RISC-V Assembly Language

August 1st 2025 (Alpha release)

Alan Johnson

Published by xelsys

Copyright © 2025 Alan Johnson. All rights reserved.

ISBN

Target Audience

- Improvers with basic C/Linux experience
- Embedded systems enthusiasts
- Computer architecture learners
- Developers interested in system-level programming
- Computer science students

Pre-requisites

Knowledge of the following areas will ease the journey.

- Familiarity with basic computer hardware

Microprocessor architecture

- Memory and data buses, register, ALUs, …
- Experience with Linux ®
    - Installation of the Operating System and applications
    - Bash
- Basic knowledge of the C programming language
- High school level mathematics[1]
- A RISC-V system[2] or an emulated device.

[1] Some of the *optional* tasks involve linear algebra, which will be more familiar to those at a higher level. A good reference is found at https://www.khanacademy.org/math/linear-algebra

[2] RISC-V based hardware is preferred over simulation.

Summary of the document

Overview

This book provides an in-depth introduction to assembly language concepts using the RISC-V instruction set architecture. It emphasizes low-level programming, number systems, memory, and architecture-specific coding techniques. The content is structured into logical chapters with practical examples, exercises, and diagrams. There may be areas that require supplemental knowledge.

Foundational Concepts (Chapters 1–2)

- **Assembly vs. High-Level Languages**: Assembly gives full control over hardware, unlike abstracted high-level languages.
- **Assembling Process**: Involves assembling, compiling, and linking.
- **Hardware, Software, Firmware**: Differentiates their roles in a system.
- **Number Systems**: Covers binary, hexadecimal, BCD, conversions, and complements.
- **Logic Operations**: Introduces AND, OR, XOR, NOT, with truth tables.
- **RISC-V Origins & Architecture**: Discusses open-source nature, 32/64-bit modes, instruction extensions (e.g., M, F, D, C).
- **Register Set**: 32 general-purpose registers with ABI names (e.g., ra, sp, a0-a7).

RISC-V Assembly language Programming Concepts (Chapters 3–6)

- **Memory Access**: Load/store instructions, addressing modes (absolute, relative).
- **Arithmetic/Logic**: Integer math, shifts, multiplication, division, condition codes.
- **Branching & Loops**: Conditional (B-type) and unconditional (J-type) branching, loop counters.
- **Stack, Macros, Functions**: Stack usage, modular code, macro definitions and examples.

Integration with C, floating-point and vector operations (Chapters 7–9)

- **C and Assembly Integration**: In-line assembly, extended ASM, calling conventions.
- **Floating-Point Arithmetic**: IEEE 754 formats (single, double), rounding, comparisons.
- **Vector Operations**: Vector registers, SIMD-style instructions, CSR manipulation, masking, merging.

Cross-compiling Tools and Simulation (Chapter 10)

- Spike Simulator & Toolchain: Installation, cross-compilation, debugging.

CONTENTS

Chapter 1. The fundamentals of assembly language. 1-1

1.1. What is assembly language? 1-1

1.1.1. High-level languages Vs Assembly language 1-1

1.1.2. Architecture and Machine code 1-1

1.1.3. Assembling, compiling and linking 1-1

1.1.4. Pseudocode 1-2

1.1.5. Why use assembly? 1-2

1.2. Hardware *Vs Software Vs Firmware* 1-2

1.2.1. Hardware 1-2

1.2.2. Software 1-3

1.3. Number Systems 1-3

1.3.1. Binary, Octal, Hexadecimal 1-3

1.3.2. Converting Hexadecimal to Decimal 1-6

1.3.3. Converting Decimal to Hexadecimal 1-6

1.3.4. Binary Fractions 1-7

1.3.5. Converting *a binary fraction to decimal* 1-7

1.3.6. One and Two's complement 1-8

1.3.7. Addition and subtraction of binary numbers 1-9

1.3.8. *Binary subtraction* 1-10

1.3.9. *Binary multiplication* 1-11

1.3.10. *Binary Division* 1-12

1.3.11. Shift/ Rotate instructions to perform multiply *and divide operations* 1-13

1.3.12. Binary Coded Decimal (BCD) 1-13

1.3.13. Floating Point 1-16

1.4. Logic operations – and, OR, Exclusive OR, NOT 1-20

Summary 1-23

Exercises for chapter1 1-24

Chapter 2. Getting Started 2-1

2.1. Origins of RISC-V 2-1

2.2. Architecture 2-1

2.2.1. RISC-V Registers 2-4

2.2.2. Additional fields funct3 and funct7 2-12
2.3. Coding Tools 2-13
2.3.1. Editing files 2-14
2.3.2. Comments 2-17
2.3.3. Assembling 2-17
2.3.4. Linker 2-18
2.3.5. GDB – The GNU Debugger 2-21
2.3.6. Objdump 2-23
2.3.7. Make 2-24
2.4. Choosing a candidate platform 2-27
2.4.1. Hardware Platforms 2-27
2.4.2. Emulation and Simulation 2-27
2.4.3. Using strace 2-39
RISC-V Instructions Covered in Chapter 2 2-40
Exercises for chapter 2 2-41
Chapter 3. Dealing with memory 3-2
3.1. Load and Store instructions 3-2
3.1.1. LOAD Instructions (Memory → Registers) 3-2
3.2. Outputting (Writing) ASCII text 3-5
3.3. Inputting (reading) values 3-6
3.4. Relative and absolute addressing 3-8
3.4.1. RISC-V Assembler Modifiers 3-8
3.5. Linker Relaxation 3-13
3.5.1. Enhancements to GDB 3-19
Exercises for chapter3 3-21
RISC-V instructions covered in chapter 3 3-22
Chapter 4. Arithmetic operations (First Pass) 4-1
4.1. Data Sizes 4-1
4.2. Integer Instructions 4-1
4.2.1. Register ADD 4-1
4.2.2. ADD Immediate 4-7
4.2.3. MV instruction 4-8
4.3. Condition Codes 4-9

4.3.1. Detecting an oVerflow condition 4-10
4.3.2. RVM Instructions 4-10
4.3.3. Multiply Instructions 4-10
4.3.4. Divide Instructions 4-14
4.3.4.1. Division by zero 4-15
4.4. Shift Operations 4-16
4.5. Logical Instructions 4-18
4.5.1. Logical function observations 4-22
Exercises for chapter 4 4-23
RISC-V instructions covered in chapter 4 4-24
Chapter 5. Loops, Branches and Conditions 5-1
5.1. J-Type and B-Type instructions 5-1
5.1.1. B-Type instruction details 5-1
5.1.2. J-Type instruction details 5-2
5.2. Implementing a loop counter to square numbers 5-3
Exercises for chapter 5 5-5
RISC-V instruction covered in chapter 5 5-6
Chapter 6. The Stack, Macros and Functions 6-1
6.1. Overview 6-1
6.1.1. The Stack 6-1
6.1.2. Combining separate programs 6-2
6.2. Macros 6-10
Exercises for chapter6 6-13
Summary of instructions used in chapter 6 6-14
Chapter 7. RISC_V assembly and C together 7-2
7.1. Calling assembly functions from a high-level language 7-2
7.1. Using in-line code 7-7
7.1.1. Basic ASM 7-7
7.1.1. Extended ASM 7-7
7.2. Optimizing code with GCC 7-11
7.3. Format Specifiers 7-14
Exercises for chapter 7 7-17
Summary of RISC-V instruction used in chapter 7 7-18

Chapter 8. Floating-Point 8-1
8.1. RISC-V floating-point capability 8-1
8.1.1. Floating-point register set 8-1
8.2. Instruction types 8-3
8.2.1. Arithmetic instructions 8-3
8.2.2. Load and store instructions 8-4
8.2.3. Convert instructions 8-4
8.2.4. Categorization instructions 8-4
8.2.5. Comparison instructions 8-4
8.2.6. Miscellaneous instructions 8-4
8.3. Instruction format 8-4
8.3.1. Floating point control and status register 8-5
8.3.2. Rounding Modes 8-5
8.3.3. Accrued Exception bits 8-6
8.4. Floating-Point comparison instructions 8-18
8.5. Floating-point classification instructions 8-18
8.6. Exercises for chapter 8 8-22
8.7. Summary of RISC-V instructions used in chapter 8 8-22
Chapter 9. Vector operations 9-1
9.1. Vector system support 9-1
9.2. Vector registers overview 9-2
9.2.1. General purpose vector registers 9-2
9.2.2. Vector CSR's 9-3
9.3. Vector addition/ subtraction example 9-6
9.3.1. Adding a vector and a scalar 9-10
9.3.2. Vector CSR content after execution of Listing 9-2 9-13
9.4. Moving elements with vslide 9-13
9.5. Grouping vector registers 9-15
9.5.1. Masking and merging 9-19
Summary of RISC-V instructions used in chapter 9 9-23
Chapter 10. Spike simulator and Cross compiling 10-1
10.1. Installing the toolchain 10-1
10.2. Installing Spike and PK 10-2

10.2.1. Spike installation 10-2
10.2.2. PK installation 10-2
10.2.3. Testing 10-2
10.2.4. Cross-compiling C code 10-3
10.2.5. Cross-assembling and linking 10-4
10.2.6. Using objdump 10-5
10.3. Further resources 10-9
Appendix A. GDB Commonly Used Commands 1
Appendix B. ASCII Code Table 1
Appendix C. References and Resources 1

FIGURES

Figure 1-1 Converting Decimal to binary using repeated division by 2_{10} 1-6

Figure 1-2 Converting Decimal to binary using repeated division by 16_{10} 1-6

Figure 1-3 Using shift operations to multiply and divide by two 1-13

Figure 1-4 Interpretation of Bias with floating point 1-18

Figure 1-5 Addition of two floating point numbers 1-20

Figure 2-1 Using lui and addi to generate a 32-bit immediate value. 2-10

Figure 2-2 CPUlator home page 2-36

Figure 2-3 Compiling and executing code with CPUlator 2-37

Figure 2-4 RARS Execution screen 2-38

Figure 2-5 Downloading RARS 2-39

Figure 3-1 AUIPC and ADDI instruction example to generate an address 3-9

Figure 3-2 GDB using TUI 3-20

Figure 4-1 ADD and ADDW instructions 4-3

Figure 4-2 Calculating LI to, 0xffdc5678 non-aliased steps 4-6

Figure 4-3 MULW instruction 4-13

Figure 4-4 SLL instruction 4-17

Figure 5-1 Breakdown of blt instruction 5-2

Figure 5-2 Bit breakdown of JAL instruction 5-2

Figure 5-3 Program flow of Makesquares listing 5-4

Figure 6-1 Stack contents operations 6-1

Figure 8-1 Floating-point registers 8-2

Figure 8-2 FADD bit fields 8-5

Figure 8-3 **FCSR** bit definitions 8-5

Figure 8-4 Field breakdown of FADD.s f2,f0,f, rtz instruction 8-5

Figure 8-5 GDB showing floating-point number classification 8-21

Figure 8-6 Annotated instruction steps to generate a subnormal number 8-21

Figure 9-1 Vtype register bit fields 9-4

Figure 9-2 Using the CSRR instruction to view Vector CSR values 9-5

Figure 9-3 Simultaneous addition of multiple array elements 9-6

Figure 9-4 GDB showing vector elements 9-8

Figure 9-5 Bit field breakdown for vector store instruction 9-10
Figure 9-6 Adding a scalar to all elements of a vector 9-10
Figure 9-7 Grouping vector registers 9-16
Figure 9-8 Loading two vector registers with one instruction 9-18
Figure 9-9 Operating on two vector registers with a single add instruction 9-19
Figure 9-10 CSR registers after execution of the vsetivli t0, 16, e32, m2 instruction 9-19

LISTINGS

Listing 2-1 Assembly code example 2-14
Listing 2-2 Interacting with assembly sections. 2-15
Listing 3-1 Basic read (load) and write (store) memory operation 3-3
Listing 3-2 Use of the Write Syscall 3-5
Listing 3-3 Input operation 3-6
Listing 3-4 PC-Relative addressing example 3-11
Listing 3-5 Using absolute addressing with %lo and %hi 3-12
Listing 3-6 Non relaxed version of code 3-13
Listing 3-7 Relaxed version of code 3-15
Listing 4-1 Basic Addition using ADD 4-1
Listing 4-2 ADDi example 4-7
Listing 4-3 MV instruction 4-8
Listing 4-4 Use of SUB and SUBW instructions 4-9
Listing 4-5 Multiply instructions on RV32 4-11
Listing 4-6 Multiply instructions on RV64 4-12
Listing 4-7 Division example 4-14
Listing 4-8 Further Division examples 4-15
Listing 4-9 Shift instructions example 4-17
Listing 4-10 Logical Instructions (RV32) 4-19
Listing 4-11 Logical Instruction (RV64) 4-20
Listing 5-1 Squaring numbers from 1 to 20 5-3
Listing 6-1 main.s 6-2
Listing 6-2 Makefile example using two programs 6-7
Listing 6-3 Macro example (callmacro.s) 6-10
Listing 6-4 called macro program (printmacro.s) 6-11
Listing 7-1 RISC-V multiply function called from C 7-2
Listing 7-2 Extended ASM example 7-8
Listing 7-3 Basic ASM example 7-13
Listing 7-4 Using the printf function with assembly code 7-15
Listing 8-1 Adding two double-precision floating-point numbers 8-6

Listing 8-2 Floating-point rounding using static modes 8-8
Listing 8-3 Using dynamic rounding mode 8-12
Listing 8-4 Use of sqrt instruction and reading the FCSR register 8-14
Listing 8-5 Classification of numbers - subnormal and quiet NaN 8-19
Listing 9-1 Vector to vector addition/subtraction 9-6
Listing 9-2 Adding a vector and a scalar 9-10
Listing 9-3 Use of vector vslide instructions 9-13
Listing 9-4 Grouping vector registers 9-16
Listing 9-5 Use of vmerge instruction 9-19

Tables

Table 1-1 Binary, Decimal and Hexadecimal equivalents 1-4
Table 1-2 Converting Binary to Decimal 1-4
Table 1-3 Converting decimal to binary 1-5
Table 1-4 Signed number representation. 1-8
Table 1-5 Signed and unsigned numbers 1-9
Table 1-6 Data type sizes 1-10
Table 1-7 Double-Dabble example 1-15
Table 1-8 Three digit double dabble example 1-16
Table 1-9 Floating-Point formats 1-17
Table 1-10 BIAS within single precision IEEE 754 1-18
Table 1-11 Truth table - AND 1-21
Table 1-12 Truth table - OR 1-21
Table 1-13 Truth table - XOR 1-21
Table 1-14 Simple example of encoding text using XOR 1-22
Table 2-1 Base integer instruction set variants 2-4
Table 2-2 RISC-V Integer Register Set 2-5
Table 2-3 Caller/Callee Responsibility for X registers 2-5
Table 2-4 Bit fields of the addi I-type instruction 2-8
Table 2-5 Bit fields of the add R-Type instruction 2-8
Table 2-6 Bit fields of the sw S-Type instruction 2-9
Table 2-7 Bit fields of the auipc U-Type instruction 2-10
Table 2-8 Bit fields of the B-Type instruction 2-11
Table 2-9 Bit fields of the J-Type instruction 2-12
Table 2-10 Funct field usage with instruction types. 2-12
Table 2-11 Funct fields used for R-Type Integer instructions 2-13
Table 2-12 GNU Tools associated with assembling and linking 2-13
Table 2-13 Assembly language sections 2-15
Table 2-14 Commonly used GDB commands 2-23
Table 3-1 Using GDB to display memory contents 3-2
Table 3-2 Parameters required by the Write syscall 3-5

Table 3-3 Parameters required by the read syscall 3-6
Table 3-4 Absolute and relative adressing 3-10
Table 3-5 Comparison of relaxed and non-relaxed code 3-17
Table 4-1 Data Types 4-1
Table 4-2 Sign extension example 4-5
Table 4-3 Detecting an overflow condition (signed) 4-10
Table 4-4 Detecting an overflow condition (unsigned) 4-10
Table 4-5 RVM Multiply Instructions 4-14
Table 4-6 RV32 Shift Instructions 4-16
Table 4-7 RISC-V Logical Instructions 4-19
Table 5-1 Conditional branch instructions 5-2
Table 7-1 Partial Stack interaction of listing7-1 7-6
Table 7-2 Inline assembly template 7-7
Table 7-3 GCC optimization options 7-12
Table 7-4 printf format specifiers 7-14
Table 8-1 Bit fields of single and double precision floating-point numbers 8-3
Table 8-2 Floating-point register width 8-3
Table 8-3 Field meaning of FADD.s instruction 8-4
Table 8-4 Rounding mode bits 8-5
Table 8-5 Floating-point comparison instructions 8-18
Table 8-6 Floating-point classes 8-18
Table 9-1 Vector CSRs 9-3
Table 9-2 Vtype SEW bit meaning 9-3
Table 9-3 LMUL and grouping correspondence 9-15
Table 10-1 Spike interactive commands for debugging 10-8

Chapter 1. The fundamentals of assembly language.

Overview of the chapter

Chapter 1 lays the foundation for understanding assembly language. The focus is on general principles which are essential prior to delving into the specifics of RISC-V. Topics include the purpose, structure, and advantages of assembly programming, and introduces the number systems and logic operations that underpin low-level code.

1.1. What is assembly language?

Assembly language is a computer language that is much closer to the operation of the computer itself; today most coding is performed using languages that are easier for humans to understand. As far as assembly language goes the coding language uses *abbreviations* to give an insight to the nature of the operation being performed. An example could be `bgt` which stands for branch if greater or less than (some condition)

1.1.1. High-level languages Vs Assembly language

Many high-level languages place a strong emphasis on abstraction, treating functions as impenetrable black boxes and hides the inner workings. Assembly language takes a different approach and allows (indeed mandates) the coder to familiarize themself with the innards of the system.

The former method is like a Rapid Application Development (RAD) methodology that works well with teams whereas the second approach often includes smaller groups with specialized knowledge. Both approaches have their place. Digital computers inherently process data in one of two states (binary) so it is essential that we understand the low -level world of one's and zero's.

1.1.2. Architecture and Machine code

Processors have different *architectures* and they each understand their own *machine code* instructions – at their very heart these instructions are combinations of binary numbers that instruct the processor how to proceed. Binary numbers are cumbersome for human operators and instead of using these numbers directly a set of *mnemonic* instructions are employed. A hypothetical example could be an instruction such as `add r1, r2,r3` which would add two numbers together that are contained in register2[3] and register3, placing the result in register1 or `add r1, r2, #45` which could add the value 45 to the value contained in register2, placing the result in register1. The corresponding native machine code (again hypothetical) could be the binary code 10101100 00010010 00101100. The *mnemonic* instructions make up the *assembly language*.

1.1.3. Assembling, compiling and linking

The role of the *assembler* (program) is to convert programmer-readable assembly instructions into the corresponding machine code instructions. The output code is termed an *object* file. Conversely a *disassembler* converts machine code instructions back into assembly language. The assembler has additional roles such as understanding a set of *directives* that can define and place data into the computer's memory locations. An

[3] Registers are low-capacity, high-speed storage elements, (typically anywhere from one to eight bytes in size) contained within the processor architecture.

example could be a set of error codes defined as textual informational messages. These messages are defined by the programmer rather than the specific processor itself. There are a number of these directives, and they will be discussed in more detail as the document progresses.

Higher-level languages use more complex translators known as *compilers* to convert to machine code. After the assembly or compilation process the object files are *linked* to form an executable program. The *linker* may act on individual or multiple files. High level language instructions do not normally have a one-to-one correspondence with the underlying machine code instructions. They are designed to be more instinctive to the programmer by providing English like keywords such as `if ..then, while,` and `print.` High level languages can be *interpretive* and translated into machine code instructions during runtime, or they can be pre-compiled before runtime into native machine-code.

1.1.4. Pseudocode

Pseudocode is used prior to writing *real, syntactically*[4] *correct* code. It outlines a set of algorithmic instructions, describing the program flow at a higher level. The benefit is to focus and plan the tasks ahead without getting too involved in low-level syntactical details, though *logic errors* may still persist. The flow is typically Algorithm → Pseudocode → Actual computer code.

Although pseudocode is not strictly defined, keywords such as IF-THEN, WHILE, GREATER THAN, . . ., are used to define program-flow.

1.1.5. Why use assembly?

Assembly language has a direct relationship with the CPU that it is running on and as a result the programs will be more compact and efficient. It is also more suited to *system-level* programming. A disadvantage is that many lines of code may be required when compared to high level languages and as a result a hybrid approach may be deployed where the bulk of the code could be written using *C* or *Python* which can pass parameters *to* and accept return values *from* a smaller section of assembly code. Portability is also an issue since the assembly language is tightly coupled with the CPU that it is running on.

In the interests of education, this book will focus more on "pure" assembly coding rather than the pragmatic hybrid approach[5].

Experienced system-level coders may wish to skip this chapter or simply skim through it and treat it as a refresher. The material discussed in *this* chapter is general and does not necessarily apply to any specific system.

1.2. Hardware *Vs Software Vs Firmware*

1.2.1. Hardware

In computer terms *hardware* refers to the physical components that make up the system. Hardware is something that can be seen and touched.

---

[4] The syntax of a language is the grammatical structure of a language. Computer languages usually have a very formal structure, with the precise order of objects used in a command strictly defined. The statement "an orange has blue skin" is *syntactically* correct but *semantically* incorrect.

[5] That is not to say that hybrid programming will be ignored within this text.

1.2.2. Software

Software refers to the actual instructions that are loaded into the computer's memory. These instructions may direct the hardware to perform certain tasks. For example, the system software is responsible for displaying the result of an operation onto a hardware output device such as a display screen or printer and for taking input from a device such as a keyboard. In general, though, software is a set of instructions that cause an operation to occur such as adding two numbers together.

1.2.2.1. Firmware

Firmware can be thought of as a set of instructions residing in hardware. This definition has become somewhat blurred as these instructions were originally loaded onto read only devices (ROMs). These devices would be physically replaced when upgraded code was required. Over time Erasable Programmable integrated circuits (IC's) were introduced, which as the name implies could be written over with new code. Today, non-volatile random-access memory (NVRAM) devices are used and can often be upgraded online without even requiring a system restart. This process is sometimes referred to as *flashing* since the underlying device is often Flash memory.

1.3. Number Systems

Anthropologists can make a claim that we count in base 10 as this is the number of digits on our hands. Other cultures have used base 60 and base 20 (possibly using both fingers and toes). These number systems are not as well suited to computer systems and today[6] base 2 and base 16 dominate when using low-level assembly programming.

1.3.1. Binary, Octal, Hexadecimal

Consider the base 10 number 4673_{10} – this breaks down into:

4 x 103
+
6 x 102
+
7 x 101
+
3 x 100
= 4000 + 600 + 70 + 3 = 4673

The use of ten (0-9) different characters along with their position represented a major advance in computation when compared to systems such as the Roman counting method.

Digital electronic systems naturally gravitate towards a two-state binary system where current either flows or it does not. These two states are represented by the symbols 0 or 1.

Each binary digit is termed a *bit(b)*. For convenience binary digits are often grouped into 8 bits termed a *Byte(B)*. Since eight bits can represent numbers ranging from 00000000 through 11111111, the decimal values translate to 0 through 255. A disadvantage of binary numbers is that a three-digit decimal number may require an equivalent of up to ten binary digits. A more compact numbering system is base 16 (hexadecimal) which treats a group of four binary numbers as a single hexadecimal number. This means that two hexadecimal

[6] Base 8 - Octal was also used on many earlier computers such as Digital Equipment Corporation's PDP family of minicomputers.

numbers will represent a single byte[7]. Hexadecimal numbers use the same symbols as decimal up to the value 9, then use the characters A through F to represent the decimal numbers ten through fifteen. The hex number 10_{16} corresponds to decimal number 16_{10}.

Table 1-1 Binary, Decimal and Hexadecimal equivalents

Binary	Decimal	Hexadecimal
0000	**0**	**0**
0001	**1**	**1**
0010	**2**	**2**
0011	**3**	**3**
0100	**4**	**4**
0101	**5**	**5**
0110	**6**	**6**
0111	**7**	**7**
1000	**8**	**8**
1001	**9**	**9**
1010	**10**	**A**
1011	**11**	**B**
1100	**12**	**C**
1101	**13**	**D**
1110	**14**	**E**
1111	**15**	**F**

1.3.1.1. Converting Binary to Decimal

Each binary digit can be converted to decimal by multiplying its value by two raised to an index where the index corresponds to the bit's position.

Table 1-2 Converting Binary to Decimal

The binary number 110101_2 then, can be converted to decimal using the following steps.

$1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 =$

$32 + 16 + 0 + 4 + 0 + 1$

$= 53_{10}$

[7] A single hexadecimal number is sometimes referred to as a nibble.

1.3.1.2. Converting Decimal to Binary

The following method breaks down a decimal number into powers of two, so to convert the number 843_{10} to its equivalent binary number –

1. First get the highest power of two contained in 843 which is 512 (2^9).
2. Subtract 512 from 843 = 331,
3. The highest power of two contained in 331 is 256 (2^8),
4. Subtract 256 from 331 to get 75,
5. The highest power of two contained in 75 is 64 (2^6),
6. Subtract 64 from 75 to get 11,
7. The highest power of two contained in 11 is 8(2^3),
8. Subtract 8 from11 to get 3,
9. The highest power of two contained in 3 is 2 (2^1),
10. Subtract from 3 to get 1,
11. The highest power of two contained in 1 is 1 (2^0),
12. Subtract 1 from 1 to get 0.

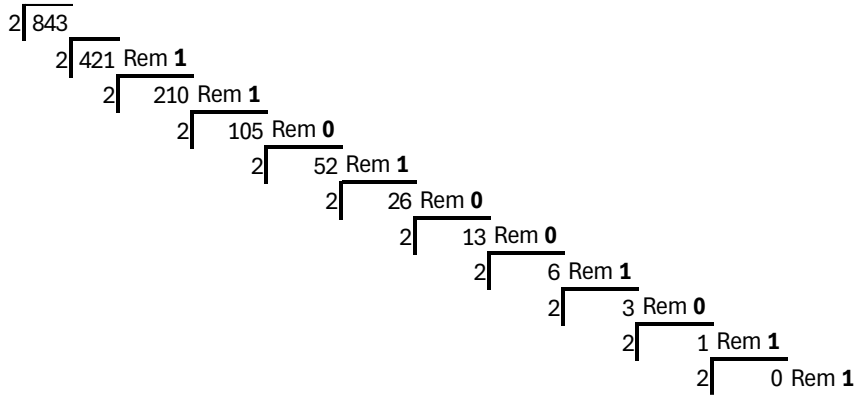
Everywhere that a power of two appears, write its index as the binary value one and where it did not appear write the binary value zero using the positional notation shown in Table 1-3.

Table 1-3 Converting decimal to binary

Value	**1**	**1**	**0**	**1**	**0**	**1**
Position	5	4	3	2	1	0
Multiply by	2^5	2^4	2^3	2^2	2^1	2^0

Another way of converting is a repeated division method. Divide the number repeatedly until zero is reached. Take note of the remainders and put the first remainder in the left-most position, then the second remainder into the left-most second position, repeating until all reminders have been recorded.

Figure 1-1 Converting Decimal to binary using repeated division by 2_{10}



Now write down the remainder starting from the *top* to get:

1101001011_2.

2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	0	1	0	1	1

1.3.2. Converting Hexadecimal to Decimal

A hex number such as $5B7C_{16}$ can be converted to decimal using a power of sixteen method –

$= 5 \times 16^3, + B \times 16^2, + 7 \times 16^1, + C \times 16^0$

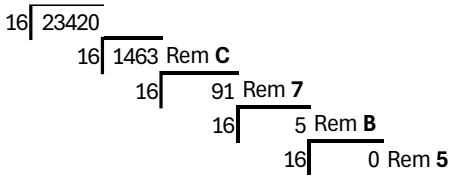
= 20,480 + 2816 + 112 + 12

= 23420

1.3.3. Converting Decimal to Hexadecimal

Take the number as shown, divide repeatedly by 16_{10} until zero is reached. Record the remainders in base 16 format (e.g. for a remainder of 10_{10}, record "A"). Note the remainders and put the last remainder in the left-most position, the second from last remainder into the left-most second position, repeating until all reminders have been recorded.

Figure 1-2 Converting Decimal to binary using repeated division by 16_{10}



Again, printing out the remainders from the bottom gives 5B7C

1.3.4. Binary Fractions

The binary numbers that have been dealt with up to this point are *natural* number equivalents (positive whole numbers). Positional notation is used to show the corresponding power of two index. [8] Fractions can be represented in binary by moving to the left of the 2^0. These values then become 2^{-1}, 2^{-2}, . . .

1.3.5. Converting *a binary fraction to decimal*

1101.01 is equivalent to the base 10 number 13.25 since we have:

$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2.}$

1.3.5.1. *Converting a decimal fraction to binary.*

Repeatedly multiply the fractional part by two until it becomes zero, taking note of the value to the left (integer portion) of the decimal point. Accumulate the values of the integer part from top to bottom to get the binary fractional part.

Example 0.625_{10}

0.625 x 2 = 1.25

0.25 x 2 = 0.5

0.5 x 2 = 1.0

Stop since the value to the right of the decimal point =0

Take the integer value from top to bottom = 0.101_2

Consider number 0.3

0.3 x 2 = 0.6

0.6 x 2 = 1.2

0.2 x 2 = 0.4

0.4 x 2 = 0.8

0.8 x 2 = 1.6

0.6 x 2 = 1.2

0.2 x 2 = 0.4

0.4 x 2 = 0.8

0.8 x 2 = 1.6

0.6 x 2 = 1.2

[8] Recall that negative indices can be resolved by changing the sign of the index and changing the operation from division to multiplication and vice versa so that $1 / 2^{-2}$ becomes $1 \times 2^2 = 4$ and $4 \times 2^2 = 4/2^{-2} = 16$

This highlighted value has been met before, so this is a recurring fraction with the pattern 0011 repeating - .0100110011… This means that when evaluating, a halt counter should be added. The logic would be to end when the fractional part = 0 or when the required degree of precision has been reached.

1.3.6. One and Two's complement

An eight-bit byte can represent any one of 256 values ranging from 0 – 255_{10}. This is known as *unsigned* notation. Another representation is to use half of the range as positive integers and the other half as negative, in this case the range is from +127[9] through -128. This method uses the *most significant bit* to represent the sign and is known as *signed* notation. The number line for an eight-bit signed number is:

-128, -127, …, 0, 1, 2, …, 127

Table 1-4 Signed number representation.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Sign bit	Magnitude Bits						

Interpreting the value of a signed number is straightforward –

The procedure is to add the corresponding powers of two of each bit's place value but leave out the sign bit. The next step is to add in the value of the sign bit. For positive numbers it makes no difference since the value of the sign bit is zero, but for negative numbers the value of the sign bit is -128.

Example

- Take the positive binary number 00101100
- Add the magnitude bits together

$0x2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$

$= 32 + 8 + 4 = 44$

- Add in the value of the sign bit (2^7) to get:-

$0 + 44 = 44$

- For the negative number 10011001
- Add the magnitude bits together.

$0x2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

$= 16 + 8 + 1 = 25$

- Add in the value of the sign bit (2^7)to get

$-128 + 25 = -103$

[9] Zero is treated as a positive number here

Converting from a signed number to an unsigned number is a simple operation, the procedure is to invert the bits and then add the binary value 1.

So, to convert the positive number 63_{10} to negative 63_{10}.

- Convert the number to an eight-bit binary number -

00111111

- Invert the bits to get -

11000000 (one's complement)

- Add 1 to get –

11000001 (Two's complement)

- Convert back to decimal to get:-

-128+64+1 = 63

The first stage of inverting the bits - obtains the one's complement, adding the binary digit 1 to the one's complement - obtains the two's complement.

The following table shows an extract of the first few signed numbers.

Table 1-5 Signed and unsigned numbers

Signed Binary Number	Decimal Equivalent
0111 1111	127
0111 1110	126
0111 1101	125
. .	.
0000 0000	0
1111 1111	-1
1111 1110	-2
.. . .	
1000 0010	-126
1000 0001	-127
1000 0000	-128

1.3.7. Addition and subtraction of binary numbers

1.3.7.1. Binary Addition

To add two binary numbers together is straightforward, there are only four outcomes.

0 + 0 = 0

0 + 1 = 1

1+ 0 = 1

1 + 1 =10 (0+ carry)

An example of an unsigned binary addition follows-

Add 0 0 1 0 1 1 0 1 to 0 1 1 1 0 1 0 0

0	**0**	**1**	**0**	**1**	**1**	**0**	**1**
0	1	1	1	0	1	0	0
1	0	1	0	0	0	0	1

Checking by adding the decimal number equivalents together –

45 + 116 = 161

Consider the case when these numbers being added are in signed notation – here adding two positive numbers together would result in a negative number since the sign bit of the result = 1. This is an *overflow* condition since the result of 161 is clearly outside of the maximum positive number that can be represented in signed eight-bit binary arithmetic and is something that needs to be checked. Often there are conditions built-in to the processor architecture to detect this kind of situation.

Larger numbers can be dealt with by using two bytes for storage, treating the second byte as having the values 2^8 through 2^{15}. Assemblers and compilers will refer to groups of bytes by designations such as *long int*, *word* etc. It is important to check the definitions.

One such definition is:

Table 1-6 Data type sizes

Unit	Width
Doubleword	64 bits
Word	32 bits
Halfword	16 bits
Byte	8 bits

Of course, it is important to specify signed or unsigned, again a definition for an unsigned integer in the programmer's documentation might be referred to as `uint`.

1.3.8. *Binary subtraction*

Binary subtraction can be dealt with using elementary rules for small numbers and then taking into account "borrows" rather than "carrys" but using the two's complement method described on page 1-8 is by far the preferred method for larger numbers.

The steps for binary subtraction are:

1. Obtain the two's complement of the *subtrahend* (the number that will be taken away)
2. Add this to the *minuend* (the number that will be subtracted from).

3. Add the two's complement of the subtrahend to the minuend.
4. If there is a carry after the addition, then drop the carry (final result is positive)
5. If there is no carry, then compute the two's complement of the result (final result is negative)

Taking a concrete example of subtracting 00100100 (36_{10}) from 00000010 (2_{10})

- Two's complement of the subtrahend

1101 1011 +1 = 1101 1100

- Add to the minuend

0	0	0	0	0	0	1	0	Minuend
1	1	0	1	1	1	0	0	Two's complement of subtrahend
1	1	0	1	1	1	1	0	

(Carry = 0)

Two's complement of the result is

00100001+1 = 00100010

The result is negative since the carry was false = -34

Another example -

- Subtract 45_{10} from 120_{10}
- Convert numbers to eight-bit binary

45_{10}= 0010 1101_2

120_{10} = 0111 1000_2

- Two's complement of 00101101

1101 0011

- Add to 0111 1000

0	1	1	1	1	0	0	0
1	1	0	1	0	0	1	1
0	1	0	0	1	0	1	1

(carry = 1)

The result is positive since the carry was zero, 01001011 = 75_{10}

1.3.9. *Binary multiplication*

The rules for multiplication of two bits are

0 x 0 = 0
0 x 1 = 0

1 x 0 =0

1 x 1 =1

Note anything multiplied by zero is of course zero.

Example multiply binary 10 (2_{10}) by 11 (3_{10})

	1	0	
	1	1	x
	1	0	
1	0		
1	1	0	

= 6_{10}

Note this is the same as decimal multiplication where we multiply by each of the digits and then add these results together.

1.3.10. *Binary Division*

The rules for division of two bits are as follows (recall that division by zero is invalid)

- 0 / 0 invalid
- 0 / 1 = 0
- 1 / 0 invalid
- 1 / 1 =1

Division example

Divide 1 1011 (Dividend) by 00111 (Divisor)

Using long division -

```
Divide 11011        by      111
             0  0  0  1  1
      111 | 1  1  0  1  1  Bring down
Subtract     1  1  1  ↓     the 1
             1  1  0  1
Subtract        1  1  1
             ---------
             1  1  0  ←  Remainder (since it is
                         too small to be divided by 111)
Check    by converting to base 10  27/7   = 3 with remainder 6
Dividend      27
Divisor        7
Quotient       3
Remainder      6
```

1.3.11. *Shift/ Rotate instructions to perform* multiply *and divide operations*

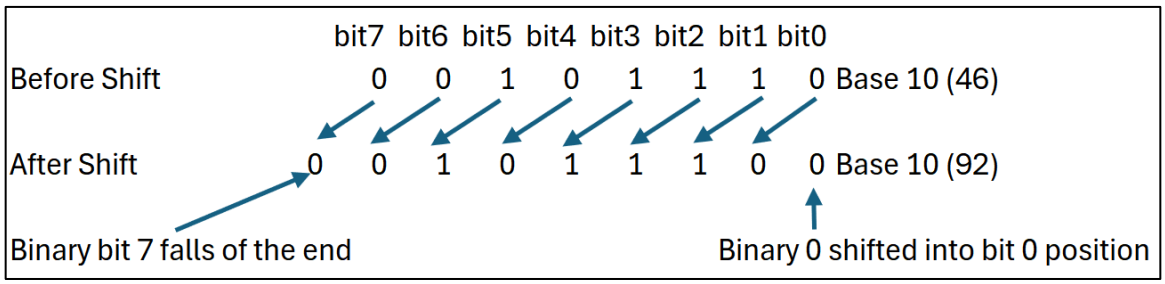
Consider an eight-bit byte 00101110 which has the decimal equivalent of 46. Next take each bit of the byte and shift them over one place to the left, filling in the now vacant bit 0 with the padded value 0 as shown below. Bit 7 has nowhere to go since it has no bit 8 position to occupy. The newly vacated bit 0 position is filled with a binary zero.

By shifting all the bits to the left the original number has been *multiplied by two* since the bit 0 value of 2^0 has been moved to the 2^1 position, bit 1's value of 2^1 has been moved to 2^2, etc.

Note that if the original bit 7 had a value of 1 then it would have been lost giving an incorrect result. This is a condition that *must* be checked for by the programmer and this will be covered in a later section.

Division by two is accomplished by shifting the bit values to the right.

Figure 1-3 Using shift operations to multiply and divide by two


bit7 bit6 bit5 bit4 bit3 bit2 bit1 bit0
Before Shift
0 0 1 0 1 1 1 0 Base 10 (46)
After Shift
0 0 1 0 1 1 1 0 0 Base 10 (92)
Binary bit 7 falls of the end
Binary 0 shifted into bit 0 position

bit 0 → bit 1→ bit 2 → bit 3 → bit 4 → bit 5 → bit 6 → bit 7 → bit 0, . . .

For simplicity the registers shown are byte-wide. In reality the width is more often 32 or 64 bits.

Other rotates are possible where the shifted-out bit feeds back to the input, giving a circular action.

Bit0 →Bit1 →Bit2 →Bit3 →Bit4 →Bit5 →Bit6 →Bit7 →Bit0 →Bit1…

1.3.12. Binary Coded Decimal (BCD)

Binary Coded Decimal represents decimal numbers in groups of bits, the encoding is normally done in four-bit nibbles. Each bit represents a power of two weight (2^3, 2^2, 2^1, 2^0, or 8,4,2,1). Since four bits can represent 16 distinct numbers, and there are only ten decimal digits, wastage occurs with this method. An alternative known as *packed BCD* may be used but is less common.

1.3.12.1. Converting Binary Coded Decimal to Decimal

BCD is similar to hexadecimal except that hex characters a through are illegal. A binary grouping of BCD characters could look like:

1001 0111 1000. Each group of 4 bits (nibbles) are read off as follows –

- 1001 = 9
- 0111 = 7
- 1000 =8

This corresponds to the decimal number 978.

1.3.12.2. BCD addition

Adding is straightforward, however if the addition of two nibbles results in a value greater than 9 (1010, 1011, 1100,1101, 1110, 1111) then it is an invalid decimal number. The resolution is to add 6 (0110) which will bring it back to a valid number. The carry will be added to the next nibble.

Addition examples –

1.

14 + 22 = 36 = 0011 0110

Verify by binary addition

0001 0100 (14)

0010 0010 (22) +

0011 0110 (36)

2.

20 +20 = 40 = 0100 0000

0010 0000 (20)

0010 0000 (20) +

0100 0000 (40)

3.

26+25 = 51 = 0101 0001

0010 0110 (26)

0010 0101 (25)+

0100 1011 Least significant nibble is greater than 9 so add 6

0000 0110 + (6)

01010001 (51)

4.

121 + 157 = 278 = 0010 0111 1000

0001 0010 0001 (121)

0001 0101 0111 (157)+

0010 0111 1000 (278)

5.

199 + 933 = 1132 = 0001 0001 0011 0010

0001 1001 1001(199)

1001 0011 0011 (933)+

1010 1100 1100 (Two nibbles invalid add 0110 0110

0000 0110 0110 +

1011 0011 0010 Now, the most significant nibble is invalid so add 6 to it

0110 0000 0000 +

0001 0001 0011 0010 (1132) Brings in a fourth nibble!

1.3.12.3. Conversion from Hex/Pure Binary to BCD

One way of converting a hex number to BCD is to convert the hex number to decimal and then to BCD. An alternative is to use the double-dabble method.

1.3.12.4. Double-Dabble

The double-dabble algorithm is fairly simple to implement; it consists of a series of shift[10] operations and additions.

Note that an n digit hex number can translate into more than n decimal digits, (8516 = 13310, FFF16 = 409510).

The method sets up a store to hold n binary digits and partitions to hold the decimal powers of two – units, tens, hundreds, thousands, … The partitions are cleared to hold all zeros and then the binary digits are shifted in one bit at a time, adjustments (addition of decimal 3) are made to the partition values dependent on their magnitude (>4). Once all bits have been shifted[11] the algorithm has been completed.

An example:

Consider the binary number 00011011 = hex 1B = decimal 27. The steps to convert from pure binary to BCD are shown in Table 1-7.

Table 1-7 Double-Dabble example

Hundreds Partition	Tens Partition	Units Partition	Binary Store	Action
0000	**0000**	**0000**	**00011011**	
0000	**0000**	**0000**	**00110110**	**Shift left-most bit over to partitions (shift1)**
0000	**0000**	**0000**	**01101100**	**Shift left-most bit over to partitions (shift2)**
0000	**0000**	**0000**	**11011000**	**Shift left-most bit over to partitions (shift3)**
0000	**0000**	**0001**	**10110000**	**Shift left-most bit over to partitions (shift4)**
0000	**0000**	**0011**	**01100000**	**Shift left-most bit over to partitions (shift5)**

[10] Shift/Rotate operations are discussed on page 1-13.

[11] The number of shifts is equal to the number of binary digits

0000	**0000**	**0110**	**11000000**	**Shift left-most bit over to partitions (shift6)**
0000	**0000**	**1001**	**11000000**	**Add 3 to units, since unit is 5 or greater**
0000	**0001**	**0011**	**10000000**	**Shift left-most bit over to partitions (shift7)**
0000	**0010**	**0111**	**00000000**	**Shift left-most bit over to partitions (shift8)**

Reading off the tens and unit columns gives the value 27_{10}.

Note 3 is added rather than 6 since the shift left operation multiplies by two!

A more complex 12-bit example is shown in Table 1-8.

Table 1-8 Three digit double dabble example

Double Dabble Three digit Hex (200) number

12 binary digits so 12 shifts are required

Hundreds	Tens	Units		Binary		
0 0 0 0	0 0 0 0	0 0 0 0	0 0 1 0	0 0 0 0	0 0 0 0	Initial State
0 0 0 0	0 0 0 0	0 0 0 0	0 1 0 0	0 0 0 0	0 0 0 0	Shift #1
0 0 0 0	0 0 0 0	0 0 0 0	1 0 0 0	0 0 0 0	0 0 0 0	Shift #2
0 0 0 0	0 0 0 0	0 0 0 1	0 0 0 0	0 0 0 0	0 0 0 0	Shift #3
0 0 0 0	0 0 0 0	0 0 1 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #4
0 0 0 0	0 0 0 0	0 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #5
0 0 0 0	0 0 0 0	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #6
0 0 0 0	0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to units
0 0 0 0	0 0 0 0	1 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	
0 0 0 0	0 0 0 1	0 1 1 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #7
0 0 0 0	0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to units
0 0 0 0	0 0 0 1	1 0 0 1	0 0 0 0	0 0 0 0	0 0 0 0	
0 0 0 0	0 0 1 1	0 0 1 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #8
0 0 0 0	0 1 1 0	0 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #9
0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to tens
0 0 0 0	1 0 0 1	0 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0	
0 0 0 1	0 0 1 0	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #10
0 0 0 0	0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to units
0 0 0 1	0 0 1 0	1 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	
0 0 1 0	1 1 0 1	0 1 1 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #11
0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to Tens
0 0 1 0	1 0 0 0	0 1 1 0	0 0 0 0	0 0 0 0	0 0 0 0	
0 0 0 0	0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to units
0 0 1 0	1 0 0 0	1 0 0 1	0 0 0 0	0 0 0 0	0 0 0 0	
0 1 0 1	0 0 0 1	0 0 1 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #12
5	1	2	200 hex = 001000000000 binary = 512 decimal			

1.3.13. Floating Point

An integer is a whole, complete and exact number such as 107 or 456. There is a limit to magnitude within a simple unit of storage such as a register. With floating -point representation a range of extremely large or

extremely small numbers can be represented at the expense of precision. This means that a floating-point number may be an approximation that introduces *rounding* to nearest digits. There are two main parts to a floating-point number, the *significand* or *mantissa* and the *exponent*. There is also provision for a sign bit. The format is *significand* multiplied by the *base* raised to a *power*, an example being 3,450,000 = 345 X 10^4. Here 345 is the significand, ten is the base and four is the exponent.

There is a standard *IEEE 754* (https://standards.ieee.org/ieee/754/6210/) which is a specification for floating-point arithmetic. The standard defines Single and Double floating-point formats[12]as shown in Table 1-9. There is also provision to include Not-a-Number[13] (NaNs) and ±Infinity.

A 32-bit single precision floating-point binary number within IEEE 754 is defined as:

Sign Bit (1 bit)	Exponent (8 bits)	Significand (23 bits)

A 64-bit double precision floating-point binary number within IEEE 754 is defined as:

Sign Bit (1 bit)	Exponent (11 bits)	Significand (52 bits)

This is summarized in Table 1-9.

Table 1-9 Floating-Point formats

Format	Bits	Significand	Unbiased Exponent	Decimal Precision
Single	32	24 [14] (23+1)	8	6-9 digits
Double	64	53 (52+1)	11	15-17 digits

1.3.13.1. Biased exponents

The use of a biased exponent can represent negative exponents. For single precision the values range from decimal +127 to -126. The bias is normally given as 2^{n-1}-1 where n is the number of exponent bits, so here we have 2^7-1= 127. The value of the biased exponent is the unbiased exponent minus 127, so that an exponent of 10011011 gives a biased exponent of (128+16+8+2+1) – 127 = 155-127 = 28.

See Table 1-10 and Figure 1-4 for more on bias.

1.3.13.2. Infinity and Not-a-number representation

A biased exponent of all ones and a significand of all zeros (-127) represents infinity. The sign bit differentiates between negative and positive infinity.

Not-a-number is represented by the biased exponent being equal to all ones (+128) and the significand being non-zero.

The sign bit is don't care.

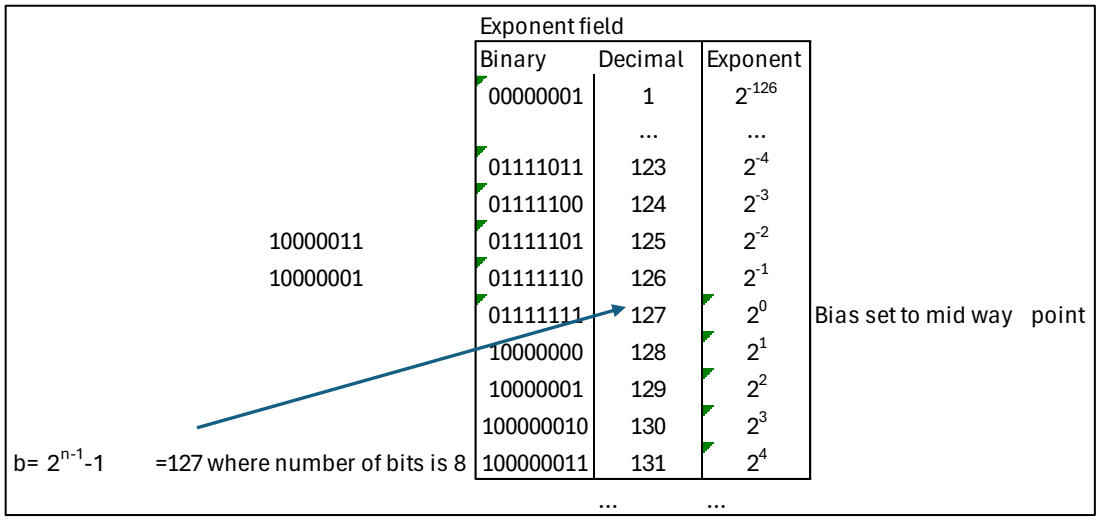
---

[12] Other formats are defined but they will not be discussed here.

[13] This could arise from operations such as divide by zero or the square root of a negative number.

[14] There is an implied bit, since the normalized format is always 1.X then there is no need to specify the "1" value to the left of the decimal point.

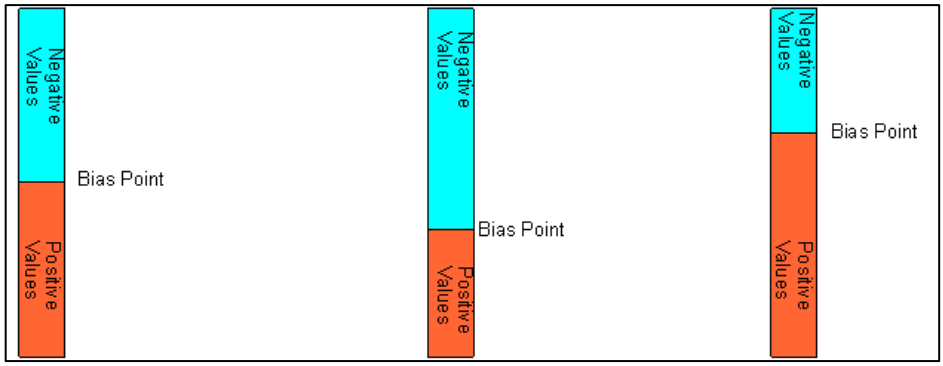
Table 1-10 BIAS within single precision IEEE 754



1.3.13.3. Understanding bias

The diagram shown in Figure 1-4 shows how varying the bias affects the ratio of negative to positive numbers. The bias used in the standard gives similar *ranges* of positive and negative exponents.

Figure 1-4 Interpretation of Bias with floating point



With double precision numbers the bias is 1023 since the unbiased component shown in Table 1-9 is 11-bits wide.

1.3.13.4. Normalized numbers

A normalized number has the form 1.XXXXX… The steps are to convert the number to binary and then perform shifts to give the desired result. Normalization shifts to the left or right depending on where the decimal point is.

Example 410.625

Steps -

1. Convert to binary (See page 1-7, if needed. for a refresher on converting decimal fractions)

= 110011010.101

2. Perform repeated shift until desired pattern us reached.

110011010.101 ÷2 (shift right operation)

1. = 11001101.0101 ÷2
2. = 1100110.10101 ÷2
3. = 110011.010101 ÷2
4. = 11001.1010101 ÷2
5. = 1100.11010101 ÷2
6. = 110.011010101 ÷2
7. = 11.0011010101 ÷2

=1.10011010101

This took a total of 8 shift operations. Add this number to 127 to get 135. Convert to binary to get:

10000111.

From our shifts earlier we had the value 10011010101, extend this to 23 bits to get 10011010101000000000000 giving the value:

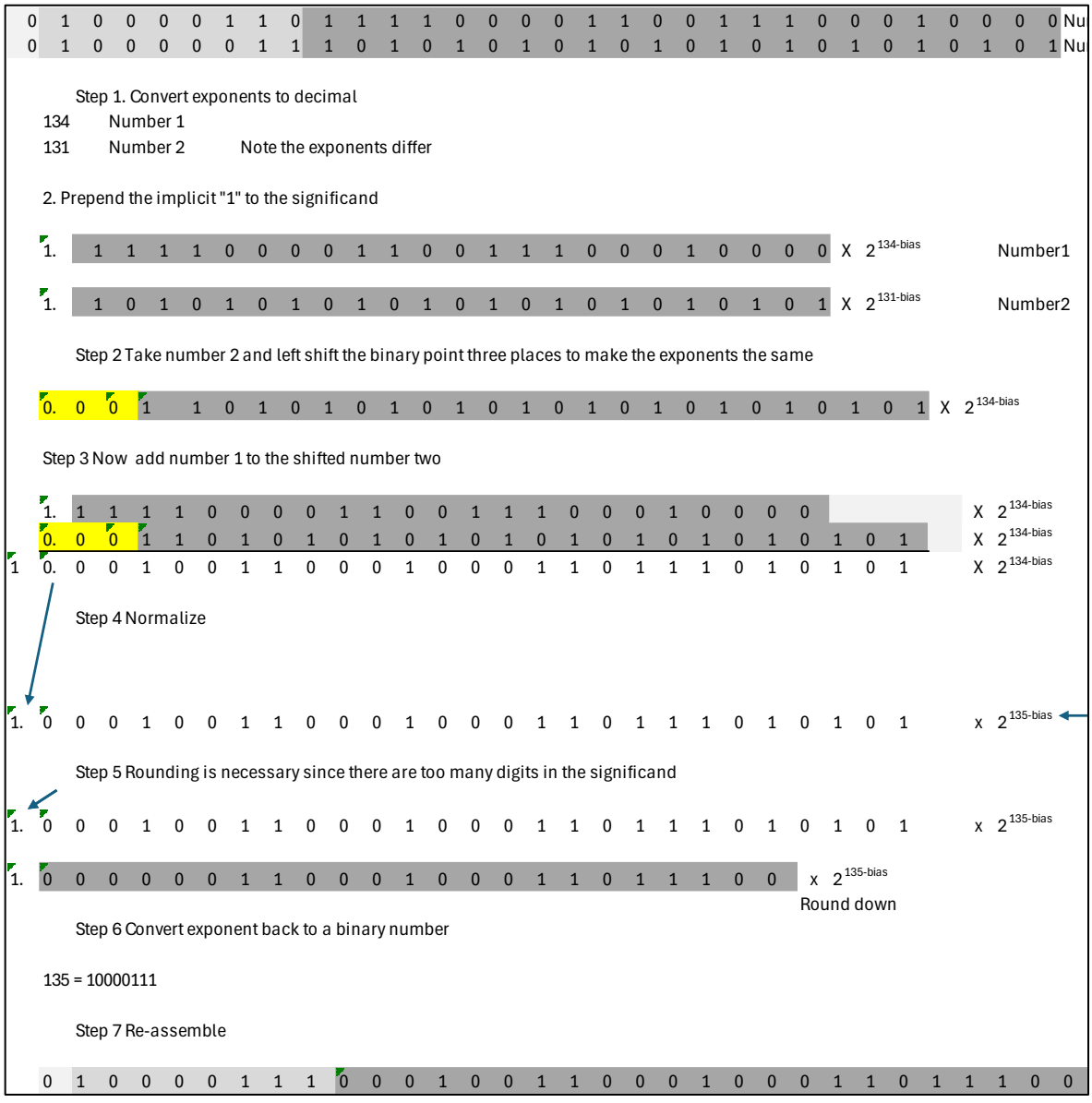
S	Exponent								Significand																		
0	1	0	0	0	0	1	1	1	1	0	0	1	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0

= 410.625

1.3.13.5. Addition of floating-point numbers

Addition is reasonably straightforward; the main concern is when the exponent differs. To equalize the exponents, take the lower number and shift over the binary point the required number of positions. So, if one exponent is 136-*Bias* and the second is 134-*Bias*, the second number needs to be shifted two places to the left.

Figure 1-5 Addition of two floating point numbers


0 1 0 0 0 0 1 1 0 1 1 1 1 0 0 0 0 1 1 0 0 1 1 1 1 0 0 0 1 0 0 0 0 Nu
0 1 0 0 0 0 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 Nu
Step 1. Convert exponents to decimal
134 Number 1
131 Number 2 Note the exponents differ
2. Prepend the implicit "1" to the significand
1. 1 1 1 1 0 0 0 0 1 1 0 0 1 1 1 0 0 0 1 0 0 0 0 X 2 134-bias Number1
1. 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 X 2 131-bias Number2
Step 2 Take number 2 and left shift the binary point three places to make the exponents the same
0. 0 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 X 2 134-bias
Step 3 Now add number 1 to the shifted number two
1. 1 1 1 1 0 0 0 0 1 1 0 0 1 1 1 0 0 0 1 0 0 0 0 X 2 134-bias
0. 0 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 X 2 134-bias
1 0. 0 0 1 0 0 0 1 1 0 0 0 1 0 0 0 1 1 0 1 1 1 0 1 0 1 0 1 X 2 134-bias
Step 4 Normalize
1. 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 1 1 0 1 1 1 0 1 0 1 0 1 x 2 135-bias
Step 5 Rounding is necessary since there are too many digits in the significand
1. 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 1 1 0 1 1 1 0 1 0 1 0 1 x 2 135-bias
1. 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 1 1 0 1 1 1 0 0 x 2 135-bias
Round down
Step 6 Convert exponent back to a binary number
135 = 10000111
Step 7 Re-assemble
0 1 0 0 0 0 1 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 1 1 0 1 1 1 0 0

1.4. Logic operations – and, OR, Exclusive OR, NOT

Logic operations are often used in decision making for example –

1. “If I feel hungry AND I have enough money, then I will order food in”.
2. “If it is cold OR it is raining, then I will wear a coat to go outside”.

3. “I can get a car discount if I pay the total amount in cash OR a I can get a lower interest rate if I take out a loan”.

Statement 1 is an AND condition and the decision to order food holds true if I am hungry AND I have enough money. Both conditions must be true.

Statement 2 is an OR condition and it states that I will wear a coat if either of these (or both) conditions are true.

Statement 3 is like statement 2 except that it is an either-or situation. Statement 2 applies equally well to both conditions in that it could be cold and also raining (similar to the AND condition). Statement 3 *exclusively* applies to the OR situation and is referred to as Exclusive OR *(XOR*).

These conditions are normally represented by *Truth Tables* such as if condition A is true AND condition B is true then result C is true. *True* and *false* values can be conveniently mapped to the binary values 1 and 0. These are known as *Boolean* variables.

Table 1-11 Truth table - AND

A	B	C
False (0)	False (0)	False (0)
True (1)	False (0)	False (0)
False (0)	True (1)	False (0)
True (1)	True (1)	True (1)

Table 1-12 Truth table - OR

A	B	C
False (0)	False (0)	False (0)
True (1)	False (0)	True (1)
False (0)	True (1)	True (1)
True (1)	True (1)	True (1)

Table 1-13 Truth table - XOR

A	B	C
False (0)	False (0)	False (0)
True (1)	False (0)	True (1)
False (0)	True (1)	True (1)
True (1)	True (1)	False (0)

Other logic functions exist such as NOT which inverts the value, so a binary zero becomes a binary one. Repeating the operation, of course, gets back to the original value. Boolean algebra is a complex topic by itself – which is dealt with in set theory.

For fun - a *simple* encoding can be done with XOR – take the word “Plaintext”, converting this to seven-bit ASCII[15] code becomes –

Table 1-14 Simple example of encoding text using XOR

Text string	ASCII code (decimal)	ASCII code (binary)	Apply XOR function with 10101010	Resultant ASCII code letter
P	80	1010000	1111010	z
l	108	1101100	1000110	.
a	97	1100001	1001011	K
i	105	1101001	1000011	C
n	110	1101110	1000100	D
t	116	1110100	1011110	^
e	101	1100101	1001111	O
x	120	1111000	1010010	4
t	116	1110100	1011110	^

So, the encoded string “Plaintext” becomes “z.KCD^O4^”.

Of course, this is easily cracked and decoded!

The following rules show the resulting bitwise values:

- X AND 0 = 0
- X AND 1 = X
- X OR 0 = X
- X OR 1 = 1

Now that the foundation is in place it is time to move from generic concepts to programming on a specific architecture!

[15] See the appendix for a table of ASCII codes

Summary

- Introduction to Assembly language
- Number Systems
- Shift Operations
- Logic and Truth tables

Exercises for chapter1

1. Convert 11.110 to base 10
2. Divide 10111101 by 111 using manual long division
3. Convert 0x1fd to BCD
4. Convert 35.65 to single precision floating-point according to IEEE 75.
5. Write a pseudocode program to convert lower case ASCII characters a-z to upper case ASCII character A-Z.
6. Convert the signed binary byte to base10
7. Convert the octal number 341 to base 16
8. What are mnemonics?
9. Describe the advantages of a high-level language over assembly language

Chapter 2. Getting Started

Overview of the chapter

Chapter 2 introduces the RISC-V architecture and essential tools required to start programming with RISC-V. It moves from theory to practical steps, providing context, tools, and setup guidance for working in RISC-V assembly.

2.1. Origins of RISC-V

The design of RISC-V uses Reduced Instruction Set Computer (RISC) architecture. RISC-V originated in 2010 as a project at the University of California, Berkeley. The suffix "V" indicates that it is the fifth generation of the RISC architecture. RISC has the advantage of a simpler design with lower power consumption making it ideal for use in embedded systems. RISC-V is now under the stewardship of RISC-V International based in Switzerland. A distinguishing feature is that it is open and royalty free.

2.2. Architecture

Implementations use a naming convention to denote which Instruction Set Architectures (ISAs) are available within a specific implementation. An example being *RV64I* or *RV32E* which stands for RISC-V with a 64-bit *integer* instruction set and RISC-V with a 32-bit reduced *integer* set respectively[16]. The integer and reduced integer designations form the *Base Integer ISA*. This is mandatory for implementations. Optional extensions are defined as:

- M for integer multiplication and division.
- A for Atomic extensions.
- F and D for single and double precision floating-point. Here the designation RV64IM would mean 64-bit with Integer and integer multiplication/division support.
- C for compressed Instructions.
- E for Embedded.
- There is also the ability to support non-standard extensions.

To show the RISC-V instruction set support under Linux, the command `cat /proc/cpuinfo` can be used –

```
cat /proc/cpuinfo

processor   : 0

hart        : 1
```

[16] RV128 definitions also exist but will not be discussed here.

```
isa             : rv64imafdc_zicntr_zicsr_zifencei_zihpm_zba_zbb

mmu             : sv39

uarch           : sifive,u74-mc

mvendorid       : 0x489

marchid                 : 0x8000000000000007

mimpid                  : 0x4210427

. . .

processor       : 3

hart            : 4

isa             : rv64imafdc_zicntr_zicsr_zifencei_zihpm_zba_zbb

mmu             : sv39

uarch           : sifive,u74-mc

mvendorid       : 0x489

marchid                 : 0x8000000000000007

mimpid                  : 0x4210427
```

This system is identified by the string `rv64imafdc` has 4 [17] CPU cores and supports (**I**)nteger, (**M**)ultiplication/division, (**A**)tomic and (**F**)single and (**D**)ouble precision floating point with the ability to handle the smaller code size of (**C**)ompressed instructions. A designation of G represents IMAFD. The architecture shown is 64-bit. The four processors (0-3) are associated with four *harts* (1-4). A hart is a *hardware thread*[18] that can execute its own set of instructions independently of the others. Usually there is a one-to-one correspondence[19] between harts and processors.

The next output is taken from a Banana Pi BPI-F3 system showing eight processors with rv64imafdc**v** support.

Note the inclusion of the V-extension which is the vector ISA extension. Vector support is an additional bonus as few systems today support vector operations.

[17] Only the first and last CPU cores are shown in the output.

18 A hardware thread is distinct from a software thread. Software threads are multiplexed tasks, controlled by techniques such as time-slicing giving the illusion of separate tasks, whereas are hardware threads are true independent execution units.

19 Hyper-threading gives the appearance of multiple cores within a processor and so could support more than one hart.

```
$ cat /proc/cpuinfo
processor       : 0
hart            : 0
model name      : Spacemit(R) X60
isa                                                                 :
rv64imafdcv_zicbom_zicboz_zicntr_zicond_zicsr_zifencei_zihintpause_zihpm_
zfh_zfhmin_zca_zcd_zba_zbb_zbc_zbs_zkt_zve32f_zve32x_zve64d_zve64f_zve64x
_zvfh_zvfhmin_zvkt_sscofpmf_sstc_svinval_svnapot_svpbmt
mmu             : sv39
uarch           : spacemit,x60
mvendorid       : 0x710
marchid         : 0x8000000058000001
mimpid          : 0x1000000049772200
. . .
processor       : 7
hart            : 7
model name      : Spacemit(R) X60
isa                                                                 :
rv64imafdcv_zicbom_zicboz_zicntr_zicond_zicsr_zifencei_zihintpause_zihpm_
zfh_zfhmin_zca_zcd_zba_zbb_zbc_zbs_zkt_zve32f_zve32x_zve64d_zve64f_zve64x
_zvfh_zvfhmin_zvkt_sscofpmf_sstc_svinval_svnapot_svpbmt
mmu             : sv39
uarch           : spacemit,x60
mvendorid       : 0x710
marchid         : 0x8000000058000001
mimpid          : 0x1000000049772200
```

Currently there are four (separate) *base* ISAs with discussion on a 128-bit (128I) implementation. A summary is shown in Table 2-1.

Table 2-1 Base integer instruction set variants

Name	Address Space/Register Width
RV32I	32-bit
RV64I	64-bit
RV32E	32-bit
RV64E	64-bit

2.2.1. RISC-V Registers

Registers are locations that store values, they are similar to variables in high-level languages.

The primary way of interfacing with the RISC-V system is via the register set. Generically the registers may be referred to as Rd (destination register), Rs1 (first source register), Rs2 (second source register). Registers are denoted by their Application Binary Interface (ABI) name to make it more convenient to the coder. This is like high level languages where variables are given meaningful descriptive names.

2.2.1.1. Register Set

There are 32^{20} unprivileged integer X registers whose width is determined by the instruction set which is either 32,64 or 128 bits as shown in Table 2-2 along with a brief description. The registers have aliased names which reflect their usage such as X1 = ra which holds the return address. The aliased names are referred to as the ABI (Application Binary Interface) register name. Even though the registers are general-purpose, the aliased name function should be respected. For example, the saved and temporary registers are used for functions where the coder knows when to save registers prior to making the call and when they do not need to.

When the programmer calls a routine, it is termed the *calling* routine and the routine that is being called is the *callee* routine. The temporary registers (t0-t6) are saved by the *caller* and the saved registers (s0 – s11) are saved by the *callee*. This responsibility is shown in Table 2-3.

It is only necessary to save the registers that are involved in the routines. So, if the caller was not using register t1 then it would not be necessary to save it prior to involving the call.

There are also 32 floating-point registers accessible to the programmer which will be discussed at a later point in the book.

The program counter (PC) keeps track of program execution and is not used as a general-purpose register. The term *XLEN* refers to the data width 32, 64 or 128 bits.

20 RV32E and RV64E have 16 registers. The non-contiguous layout is for consistency between register sets

Table 2-2 RISC-V Integer Register Set

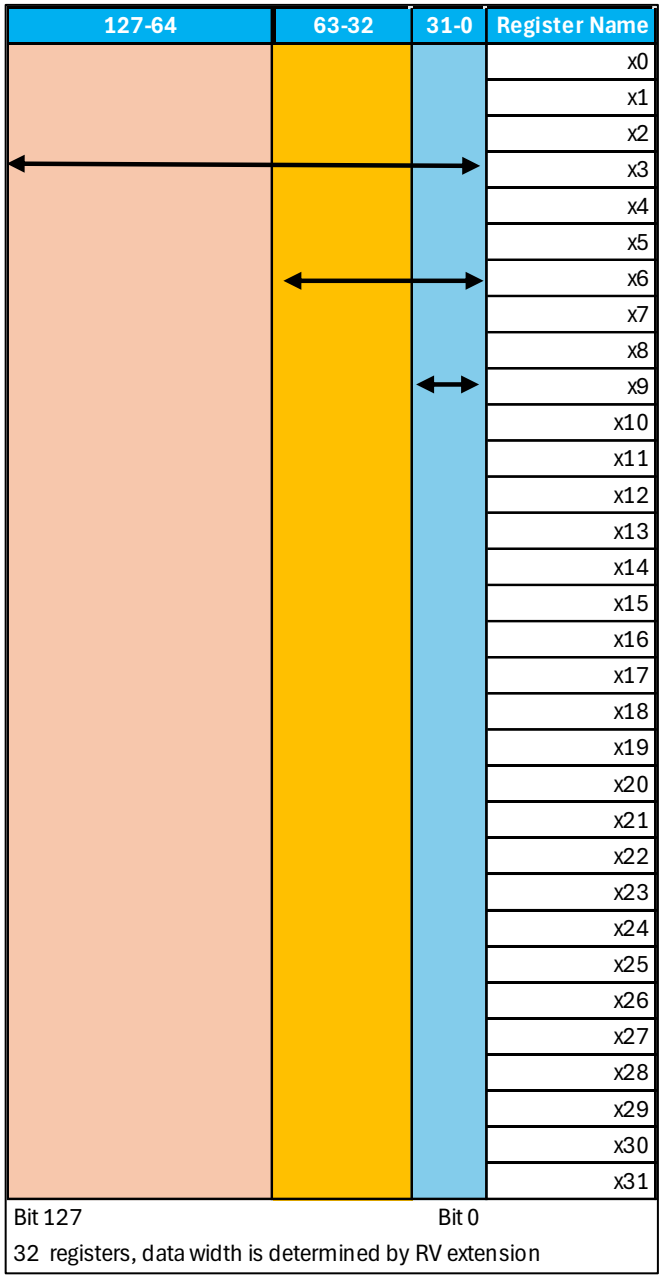


Table 2-3 Caller/Callee Responsibility for X registers

Register Name	Aliased ABI Name	Saver responsibility
x0	zero	N/A
x1	ra	Caller
x2	sp	Callee

x3	gp	N/A
x4	tp	N/A
x5	t0	Caller
x6	t1	Caller
x7	t2	Caller
x8	s0/fp	Callee
x9	s1	Callee
x10	a0	Caller
x11	a1	Caller
x12	a2	Caller
x13	a3	Caller
x14	a4	Caller
x15	a5	Caller
x16	a6	Caller
x17	a7	Caller
x18	s2	Callee
x19	s3	Callee
x20	s4	Callee
x21	s5	Callee
x22	s6	Callee
x23	s7	Callee
x24	s8	Callee
x25	s9	Callee
x26	s10	Callee
x27	s11	Callee
x28	t3	Caller
x29	t4	Caller
x30	t5	Caller
x31	t6	Caller

2.2.1.2. RV32I Base Instruction Set

The instructions are 32 bits wide; the general format is to include common fields such as:

Field Name	Role
rd	Destination Register
rs1	Source Register 1
rs2	Source Register 2
opcode	Operation Code

func	3 bit (funct3) and 7 bit (funct7) defines a particular operation
imm	Constant value such as 0x5F

2.2.1.3. Base instruction Formats

There are 4 Base Instruction Formats known as"

- I-type
- R-type
- S-type
- U-type

In addition, there are two variants of the I-type instruction known as *B-type* and *J-type*. These instructions use conditional and unconditional branches respectively to alter program flow. The immediate fields are used to encode the branch destination.

Conditional branches depend on whether certain program events have occurred, an example could be a *decrementing counter*, where a branch in the program logic only occurs if the counter has reached value zero.

An unconditional branch happens regardless of conditions. An example might be a jump to an *interrupt handler service routine*[21] if a critical or non-critical event was encountered.

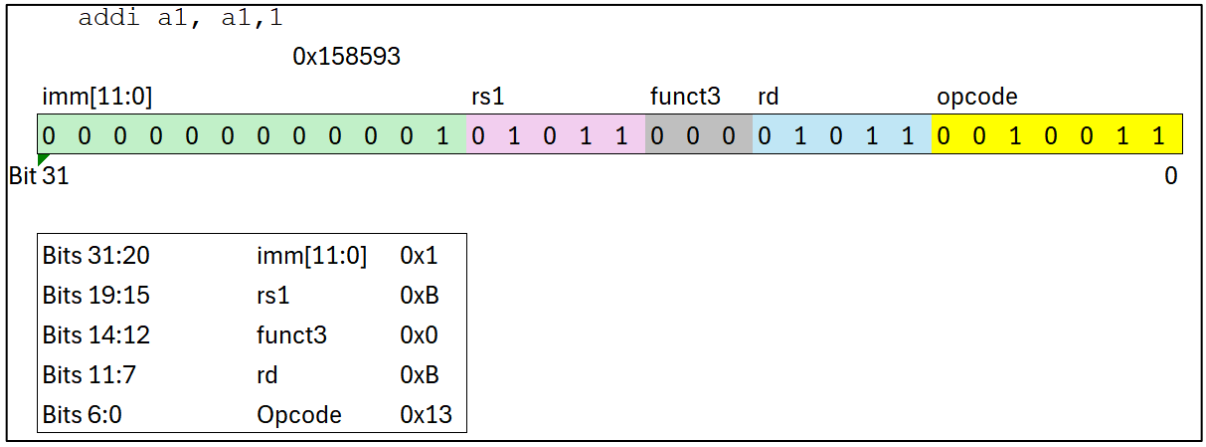
2.2.1.3.1. I-type Instruction

Most of the fields occupy the same bit positions across instructions. An example is the instruction `addi a1,a1,1`, which disassembles to 0x00158593. The breakdown of the fields for this instruction is shown in Table 2-4.

The `addi` instruction format is termed *I-type* for immediate. The instruction adds the contents of register a1 plus a constant of 1 to register a1, so the effect is to increase the value of a1 by 1. The a1 register in both the rd and rs1 fields corresponds to the value 0xb which is the 11th X register, so although the programmer uses the more friendly register name, the machine code uses the x register number.

[21] An interrupt is normally encountered in system level programming and could be used to process an attempt to access kernel memory or a user level action such as a mouse click.

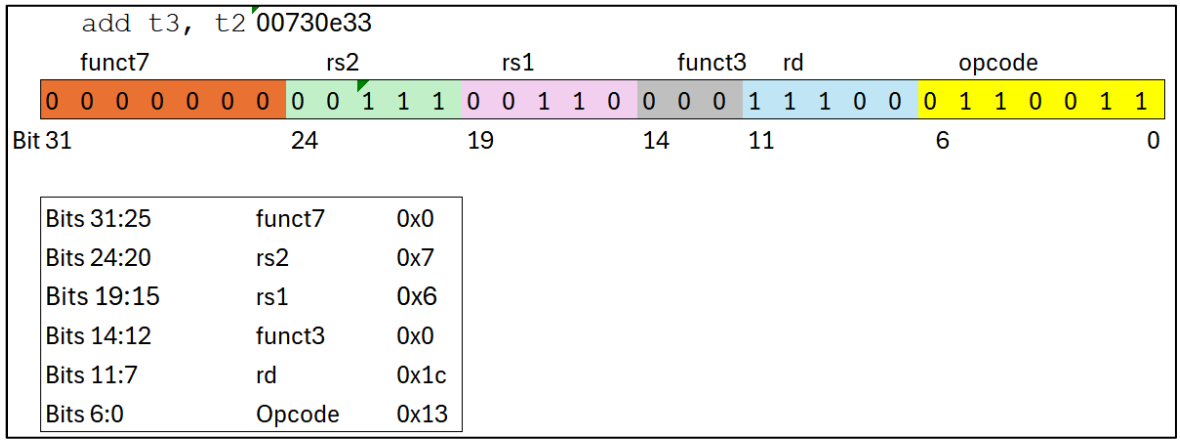
Table 2-4 Bit fields of the addi I-type instruction



2.2.1.3.2. R-type Instruction

The next instruction `add t3, t1, t2` is an *R-Type* instruction as it uses the registers for both operands. Register t2 is added to register t1 and the result is placed in register t3. Disassembly produces the machine code 0x00730e33. The field breakdown is shown in Table 2-5

Table 2-5 Bit fields of the add R-Type instruction

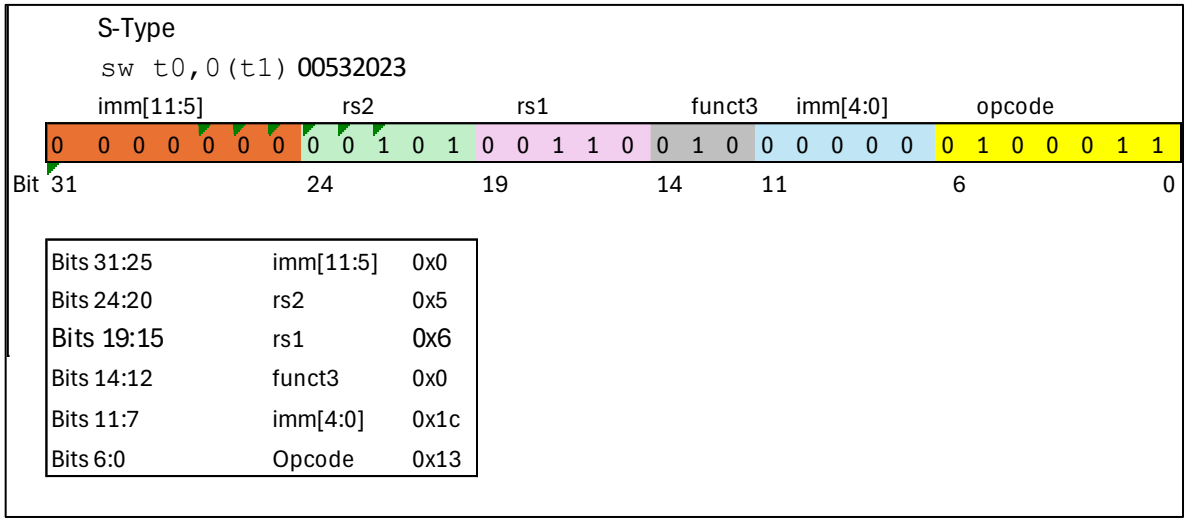


2.2.1.3.3. S-type Instruction

The Store Word instruction (`sw`) uses a register and an offset to calculate the destination address. This is an *S-type* instruction. No destination register is involved since the destination is memory. The instruction stores the 32-bits held in register t0 into the memory location pointed to by register t1 plus an immediate offset of 0. The format of the S-Type instruction is shown in Table 2-6. Note that the immediate data is broken down into two separate fields with the lower 5 bits replacing the unused (in this type of instruction) destination (rd) field.

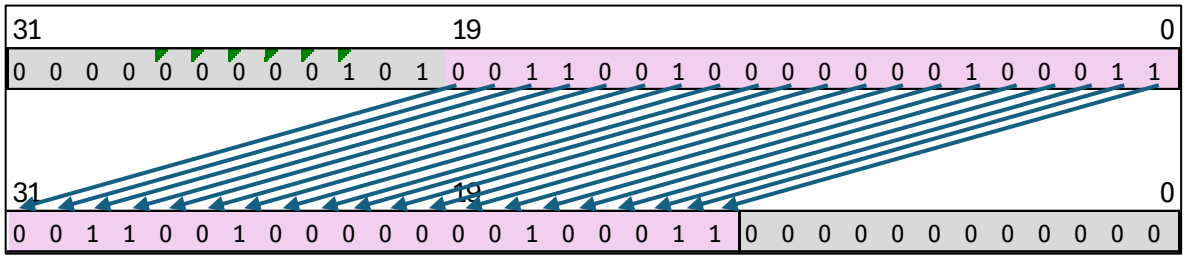
Disassembly produces the machine code 0x00532023.

Table 2-6 Bit fields of the sw S-Type instruction



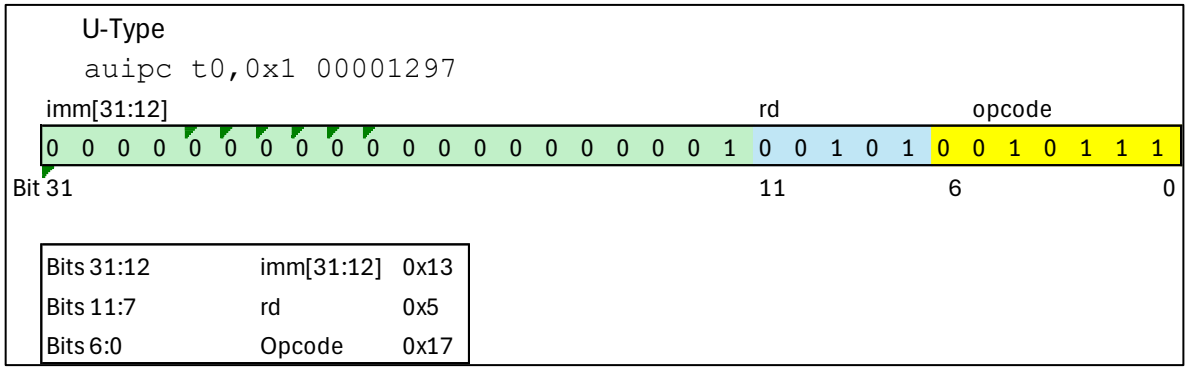
2.2.1.3.4. U-type Instruction

The *U-Type* format is used by two instructions – `lui` and `auipc`. There are 20 bits in the immediate field which permits a larger range of immediate data. These 20 bits represent immediate bits 31 through 12 with bits 11 through 0 having the value of zero. It is possible to visualize this as loading bits 19:0 and then shifting the bits 12 places to the left. This instruction combined with an instruction that supports 12-bit immediate values can be used to form a full 32-bit value.



Assume that the Program counter has the value 0x000100e8, the instruction `auipc t0,0x1` will add the immediate data 0x00001000 to 0x000100e8 giving 000110e8, placing this result in register t0. The format is shown in Table 2-7 below.

Table 2-7 Bit fields of the auipc U-Type instruction

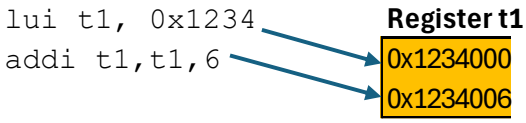


The instructions to load a 32-bit value such as 0x1234006 into a register would look like:

```
lui t1, 0x1234     # Loads upper 20 bits

addi t1,t1, 6            # Loads lower 12 bits
```

The first instruction shifts the 20-bit value over 12 places and places zeros in the lower 12 bits. The second instruction adds the 12-bit value 0x006 to the current contents of t1 (0x1234000 + 006 = 0x1234006) and places the result in t1 as shown in Figure 2-1.

Figure 2-1 Using lui and addi to generate a 32-bit immediate value.



There is however, an easier method by using the pseudo instruction - Load Immediate `li`. Instead of using the two instructions shown above, the code using `li` looks like:

```
li ti, 0x1234006.
```

This pseudo instruction could be translated into:

```
lui t1, 01234

addw t1, t1, 6
```

Pseudo instructions are automatically translated by the assembler to one or more real machine instructions.

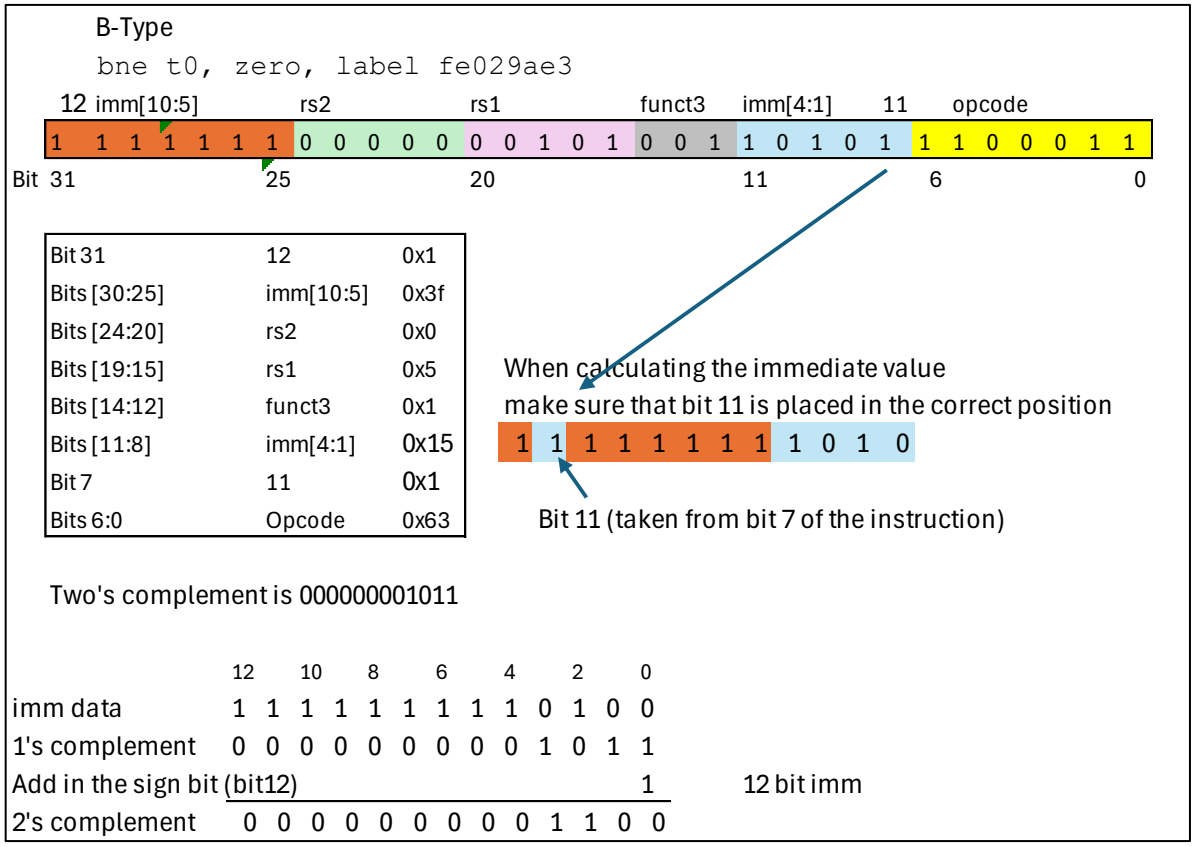
2.2.1.3.5. B-type Instruction

The *B-Type* instruction is used with *conditional branches*. With this instruction the immediate field has a range of 13-bits. This is achieved by setting the least significant bit to be zero and then substituting its bit position with bit 11 in the immediate field imm[4:1].

The location of the label `<pushit>` is at address 0x100f8 and the branch instruction is located at address 0x10104.

```
10104:       fe029ae3                 bne      t0,zero,100f8 <putit>
```

Table 2-8 Bit fields of the B-Type instruction


B-Type
bne t0, zero, label fe029ae3
12 imm[10:5] rs2 rs1 funct3 imm[4:1] 11 opcode
1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 0 1 0 0 1 0 1 0 1 1 1 0 0 0 1 1
Bit 31 25 20 11 6 0
Bit 31 12 0x1
Bits [30:25] imm[10:5] 0x3f
Bits [24:20] rs2 0x0
Bits [19:15] rs1 0x5
Bits [14:12] funct3 0x1
Bits [11:8] imm[4:1] 0x15
Bit 7 11 0x1
Bits 6:0 Opcode 0x63
When calculating the immediate value
make sure that bit 11 is placed in the correct position
1 1 1 1 1 1 1 1 1 0 1 0
Bit 11 (taken from bit 7 of the instruction)
Two's complement is 000000001011
12 10 8 6 4 2 0
imm data 1 1 1 1 1 1 1 1 1 0 1 0 0
1's complement 0 0 0 0 0 0 0 0 0 1 0 1 1
Add in the sign bit (bit12) 1
2's complement 0 0 0 0 0 0 0 0 0 1 1 0 0
12 bit imm

Since this a backwards pointing branch the immediate field is a minus value; converting it using two's complement gives a value of 0xc or 12 places back.

2.2.1.3.6. J-type Instruction

Unconditional branches use the Jump and Link instruction (`jal`). This is a *J-type* instruction. It is similar to the B-type instruction with the immediate bits in disjoint fields to allow for more efficient decoding. In this example a `jal` instruction is encountered at program counter address 0x100c0 and the instruction points to a label located at 0x100d4. The number of bytes to jump is encoded in the immediate bits as shown in Table 2-9. The machine code produced is 0x014000ef. The opcode for the instruction is 0x6f and the destination register is x1 which is the return address register (`ra`).

There is an aliased instruction `j` which uses the zero register instead of the ra register as shown below:

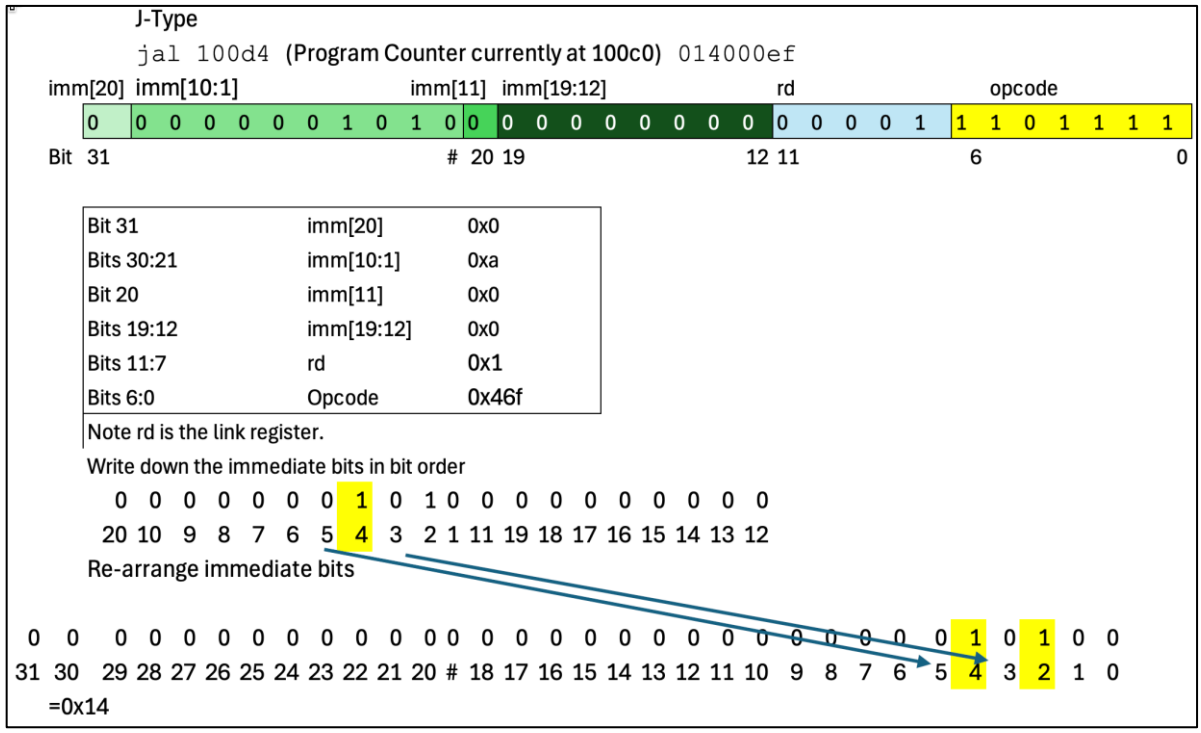
The aliased instruction `j`

```
j      100b8 <quit>
```

is encoded as:

```
jal    zero,100b8 <quit>
```

Table 2-9 Bit fields of the J-Type instruction



2.2.2. Additional fields funct3 and funct7

The opcode can be the same for different *related* instructions. The opcode is 7-bits wide occupying bit positions 6:0. To take a specific example the opcode 0110011 (0x36) refers to R-Type integer arithmetic and logical instructions. Each of these instructions are differentiated by the funct3 and funct7 fields. The numerical suffix refers to the number of bits used – funct3 is a three-bit field and funct7 is a seven-bit field. This is really a ten-bit field broken up into two sub fields. It is non-contiguous to ensure consistency across the different types of instructions (aiding decoders) and to avoid waste by reusing the bits for a different purposes, if a funct field is not required for that particular instruction type. The type of instruction defines which funct field(s) is in use. The funct3 field will always occupy bits 14:12 on any instruction that uses the field and funct7 (used on the R-Type instruction) will occupy bit positions 31:25. The U and J-type instructions do not use the funct fields and will use these bits to specify a greater range of immediate bits. Table 2-10 shows which of the funct fields are used with each instruction type.

Table 2-10 Funct field usage with instruction types.

Instruction Type	Funct3 field	Funct7 field
R-Type	Yes	Yes
I-Type	Yes	No
S-Type	Yes	No
B-Type	Yes	No

U-Type	No	No
J-Type	No	No

For R-Type instructions, the funct fields define the operation type. Some of these definitions are listed in Table 2-11 below.

Table 2-11 Funct fields used for R-Type Integer instructions

Instruction	Opcode	Funct7	Funct3
ADD	0110011	0000000	000
SUB	0110011	0100000	000
SRA	0110011	0100000	101
SLL	0110011	0000000	001
SRL	0110011	0000000	101
AND	0110011	0000000	111
OR	0110011	0000000	110
XOR	0110011	0000000	100

Note that ADD and SUB have the same funct3 value, they are differentiated by bit 5 of funct7 and similarly with SRL and SRA instructions.

2.3. Coding Tools

Chapter One gave a brief introduction to the assembly process. The tool that will be used for *assembly* is the GNU assembler (GAS). This utility is also used when compiling higher-level languages to provide intermediate code during the compilation process. It is part of the open-source GNU Binutils[22] collection. The binary tools include (amongst others) –

Table 2-12 GNU Tools associated with assembling and linking

Tool Name	Function
as	Assembler
ld	Linker
GDB	GNU Debugger
objdump	Disassembles and dumps object file information
make	Utility for assembling and linking multiple files, ignoring files that are up to date.

The candidate platforms suggested in this chapter include the tools listed in Table 2-12. The GNU tools are applicable to a wide range of architectures including Intel® and Arm®. The listings in this book have all been

[22] Use sudo apt install -y binutils - See https://www.gnu.org/software/binutils/ for more detail

tested with the GNU assembler[23]. The GNU *toolchain*[24] also includes other programming tools such as GNU Autotools and Bison for *parsing*.

The next section illustrates the use of all the tools listed in Table 2-12

2.3.1. Editing files

The first stage in the assembly process is to edit the *source* files. The assembly code is *plaintext* so basic text editors such as `vi` or `nano` should be used. By convention the file suffix is ".s", so a command to write a source program could be a command such as `vi testprogram.s` which would edit an existing file or create a new one if it did not already exist. An example of a small assembly program is shown in Listing 2-1.

Listing 2-1 Assembly code example

```
.section .text

.global _start

_start:

addi t1, zero,6   # mov 6 into t1

addi t2, zero, 11 # mov 11 into t2

add t3, t1, t2    # add t2 and t1 result goes to t3

addi a7, x0, 93   # Call

ecall
```

The first line of code defines a label (`_start)` that marks the entry point of the program. The entry `.global`[25] is a *directive* to the assembler defining an action. Directives are not part of the actual machine code that will be produced but will help the assembly process by providing instructions on how to control flow, define *symbols* and reserve space as well as other tasks. They also aid the coder in that they can define strings of text without having to refer to tables of *ASCII* codes. The code after the `_start` label is the actual code that will be assembled into RISC-V machine code.

The program moves two numbers 6 and 11 into two registers (t1 and t2) and then adds them together, placing the result of the addition in register t3. The next two lines of code use Operating System calls (*syscalls*) to gracefully exit the program.

2.3.1.1. System calls

System Calls (Syscalls) are service requests sent to the Operating Systems' kernel to perform a privileged task. These tasks include interaction with the hardware, file operations, memory management and networking. When a syscall is invoked the system switches to a *privileged* mode which executes tasks in a coordinated,

[23] The GNU assembler is recommended for all the listings here.

[24] A toolchain is a collection of programming tools.

[25] Some listings may use ".globl" . Both forms are acceptable to the GNU assembler

standard manner. Syscalls are different across architectures and Operating Systems. With RISC-V systems running under Linux, the first step is to place the syscall code into register a7 and then use the `ecall` instruction to request the function. The last two instructions of Listing 2-1 shows how to use the *exit* system call.

2.3.1.1.1. Bare Metal Programming

Bare metal programming is a term used when the code interacts directly with the machine itself, it does not have the benefit of the Operating System support and so syscalls are unavailable. Bare metal programming is commonly used in *embedded* systems.

2.3.1.2. Sections

Assembly language source files are typically divided into *sections*. The sections used with RISC-V assembly files include the following.

Table 2-13 Assembly language sections

Section Name	Purpose
.text	Contains the source level instructions.
.data	Allocation of initialized variables.
.rodata	Holds constants or text strings that are read only.
.bss	Allocates uninitialized data buffers.

The following listing shows the use of sections and how they are interacted with by registers.

Listing 2-2 Interacting with assembly sections.

```
.section .text

# The .text section contains the assembly language source instructions,

omitting

# the prefix .section also works for .text, .data and .bss sections.

.global _start

_start:

/* This program illustrates the use of sections in RISC-V assembly.

It also shows how to interact with memory via load and store instructions

Note this text is encapsulated using a multi-line comment.

The other comments using the # character are single-line comments */


# Interacting with .data section
```

```
lw a0, oneword    # Loads the content of the address at oneword into a0
lh a1, onehalf  # Loads the content of the address at onehalf into a1
lb a2, onebyte  # Loads the content of the address at onebyte into a2
# Interacting with .bss section
la a3, buffer1
sw a0, 0(a3)
addi a7, x0, 93
ecall
# Interacting with .rodata section
lb a4, min
lb a5, max
.data       # This section initializes variables.
oneword:    .word 2
onehalf:    .half 0xaa55
onebyte:    .byte 0x44
.section .rodata  # This section can hold constants and text strings
min:  .byte 32
max:  .byte 100
.bss        # This section allocates memory space for storage
buffer1: .space 100
```

Sections can be shown with the `objdump` command (see page 2-23).

```
objdump -h listing3-3b


listing3-3b:     file format elf64-littleriscv


Sections:
```

```
Idx Name      Size      VMA           LMA           File off  Algn

0 .text       00000080  00000000000100e8  00000000000100e8  000000e8  2**2

CONTENTS, ALLOC, LOAD, READONLY, CODE

1 .data           0000002d  0000000000011168  0000000000011168  00000168

2**0

CONTENTS, ALLOC, LOAD, DATA

2 .riscv.attributes 00000037  0000000000000000  0000000000000000  00000195

2**0

CONTENTS, READONLY

. . .
```

2.3.1.2.1. Virtual Memory Address and Load Memory Address

In the output of the objdump command given above there are field headings *VMA* and *LMA*. These are the Virtual and Load memory addresses. The virtual memory address is the section address at runtime and the load memory address is the location where the section is loaded.

These locations are usually the same. but can differ if ROM memory (LMA) needs to be re-located to writable RAM memory (VMA).

2.3.2. Comments

Comments are ignored by the assembler but important for maintaining code clarity. There are multi-line comments beginning with `/*` and ending with `*/` and single line comments using the `#` character.

2.3.3. Assembling

Note high-level languages have an additional stage between editing and assembling – this is the *compilation* stage which will generate assembly code from a high-level language source code.

Later in[26] the book, mixing of hybrid high-level languages and assembly code will be covered but until then, only pure assembly language programming will be discussed.

Once the file has been edited it can be assembled. The assembler will check for syntax[27] errors and if successful it will generate an object file. This is the main task of the assembler – *to generate machine code for the underlying processor architecture*. It is also responsible for translating RISC-V pseudo instructions into real machine code instructions. The object file uses the suffix ".o". The **G**NU assembler may be referred to as GAS!

---

[26] See Page 7-1.

[27] Note logic/flow error checking is largely the coder's responsibility.

The command to assemble a program is shown below:

```
as -o testprogram.o testprogram.s
```

Note the order of files where the object file name is given first followed by the source name.

When initially developing programs, it is normal to include extra information to assist with the debugging process. Once the code is ready for final release this extra information is removed. The command to include debugging information is:

```
as -g -o testprogram.o testprogram.s
```

Including the debugging data increases the size of the code. The assembler ignores the comments which are only used for human clarification purposes and have no meaning to the processor. Once the code has been translated into machine code it is not yet in an executable state. Along with the actual machine code, a number of *symbol* references may be defined. These may be references to symbols defined in other object files that the current source program has no access to.

2.3.4. Linker

The linker's role is to produce code that can be executed by the system, most large programs are not standalone but instead consist of a number of smaller programs or library files. The linker "joins" these programs together and generates the final executable. In addition, the linker has the responsibility of resolving the symbol references. It will also perform optimization.

Other considerations are integration within the file system. Linux programs use the *Executable and Linkable format* (ELF). This format is portable, supporting a wide range of platforms. ELF files consist of headers and sections to aid with mapping the program into memory. The `readelf` utility analyzes the ELF format. The ELF header can be shown with the command `readelf -h testprogram` as shown below:

```
readelf -h testprogram

ELF Header:

Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00

Class:                             ELF64

Data:                              2's complement, little endian

Version:                           1 (current)

OS/ABI:                            UNIX - System V

ABI Version:                       0

Type:                              EXEC (Executable file)

Machine:                           RISC-V
```

```
Version:                           0x1

Entry point address:               0x100b0

Start of program headers:          64 (bytes into file)

Start of section headers:          1192 (bytes into file)

Flags:                             0x4, double-float ABI

Size of this header:               64 (bytes)

Size of program headers:           56 (bytes)

Number of program headers:         2

Size of section headers:           64 (bytes)

Number of section headers:         11

Section header string table index: 10
```

To produce an executable from the `testprogram.o` file use the command –

```
ld -o testprogram testprogram.o
```

2.3.4.1. Linker Scripts

Linker scripts are used to describe memory allocation maps and are more commonly used in embedded systems. They are text files.

The command `ld -verbose` lists the contents of the default linker script –

```
GNU ld (GNU Binutils for Debian) 2.40

Supported emulations:

elf64lriscv

elf64lriscv_lp64f

elf64lriscv_lp64

elf32lriscv

elf32lriscv_ilp32f

elf32lriscv_ilp32

elf64briscv
```

```
elf64briscv_lp64f

elf64briscv_lp64

elf32briscv

elf32briscv_ilp32f

elf32briscv_ilp32

using internal linker script:

==================================================

/* Script for -z combreloc */

/* Copyright (C) 2014-2023 Free Software Foundation, Inc.

Copying and distribution of this script, with or without modification,

. . .

/* DWARF 3.  */

.debug_pubtypes 0 : { *(.debug_pubtypes) }

.debug_ranges   0 : { *(.debug_ranges) }

/* DWARF 5.  */

.debug_addr     0 : { *(.debug_addr) }

.debug_line_str 0 : { *(.debug_line_str) }

.debug_loclists 0 : { *(.debug_loclists) }

.debug_macro    0 : { *(.debug_macro) }

.debug_names    0 : { *(.debug_names) }

.debug_rnglists 0 : { *(.debug_rnglists) }

.debug_str_offsets 0 : { *(.debug_str_offsets) }

.debug_sup      0 : { *(.debug_sup) }

.gnu.attributes 0 : { KEEP (*(.gnu.attributes)) }

/DISCARD/ : { *(.note.GNU-stack) *(.gnu_debuglink) *(.gnu.lto_*) }
```

On the Debian system used here (`Linux starfive 6.6.20-starfive #41SF SMP Fri Sep 20 17:48:26 CST 2024 riscv64 GNU/Linux`) the linker scripts are located at `/lib/riscv64-linux-gnu/ldscripts/ -`

```
lf32briscv_ilp32f.x             elf32briscv_ilp32.xdc       elf32briscv.xe

elf32lriscv_ilp32f.xsceelf32lriscv_ilp32.xw          elf64briscv_lp64f.xce

elf64briscv_lp64.xdw    elf64briscv.xs              elf64lriscv_lp64f.xswe

elf64lriscv.xbn

elf32briscv_ilp32f.xbn   elf32briscv_ilp32.xdce  elf32briscv.xn

. . .
```

2.3.5. GDB – The GNU Debugger

GDB is used to view program flow. The code can be run "one step (instruction) at a time" as a teaching tool to promote understanding of program execution. It does this by displaying register and memory contents along with the line of source code being executed. The tool is invaluable by experienced coders looking to track down more elusive issues such as unexpected results. By single stepping through the code, the exact location where the error occurs can be readily identified. As mentioned earlier, the assembler `as` uses the `-g` switch to generate debugging information. The debugger can be launched by the `gdb` command.

To illustrate `GDB` in action issue the command

```
gdb testprogram
```

```
$ gdb testprogram

GNU GDB (Debian 13.2-1) 13.2

Copyright (C) 2023 Free Software Foundation, Inc.

. . .

For help, type "help".

Type "apropos word" to search for commands related to "word"...

Reading symbols from testprogram...

(gdb) list

.section .text

.global _start
```

```
start:

addi t1, zero,6   # mov 6 into t1

addi t2, zero, 11 # mov 11 into t2

add t3, t1, t2    # add t2 and t1 result goes to t3

addi a7, x0, 93

ecall

(gdb) b 1

Breakpoint 1 at 0x100b0: file testprogram.s, line 6.

(gdb) run

Starting program: /home/alan/asm/misc/testprogram

Breakpoint 1, _start () at testprogram.s:6

6           addi t1, zero,6   # mov 6 into t1

(gdb) n

7           addi t2, zero, 11 # mov 11 into t2

(gdb) n

8           add t3, t1, t2    # add t2 and t1 result goes to t3

(gdb) n

9           addi a7, x0, 93

(gdb) i r t1

t1             0x6      6

(gdb) i r t2

t2             0xb      11

(gdb) i r t3

t3             0x11     17

(gdb) q

A debugging session is active.
```

```
Inferior 1 [process 8652] will be killed.


Quit anyway? (y or n) y
```

Table 2-14 lists some of the more commonly used GDB commands

Table 2-14 Commonly used GDB commands

Command	Meaning
List	List the source assembly file
B 1	Sets a stopping point known as a breakpoint once the program runs, a number can be used or a label such as b _start
Run	Starts the program and halts at the first breakpoint (if any has been set).
N(ext)	Advances to the next (n)line(s)[28] of code, by-passing sub-routines.
S(tep)	Steps to the next (n)line(s) of code, entering sub-routines.
I(nfo) t1	Shows the contents of register t1
I(nfo) t2	Shows the contents of register t2
I(nfo) t3	Shows the contents of register t3
Q(uit)	Exits the program

GDB will be covered in more detail as the document progresses, in addition it will function as the primary learning tool to illustrate program flow and how each of the instructions work[29].

2.3.6. Objdump

The objdump utility is helpful with reverse engineering and understanding object code. The code can be disassembled to show the original source instructions using the -d switch, for example

```
$ objdump -d testprogram

testprogram:     file format elf64-littleriscv

Disassembly of section .text:

00000000000100b0 <_start>:

100b0:       00800313                li      t1,8
```

[28] Default is one line

[29] The reader is encouraged to use GDB to step through the program listings.

```
100b4:       00b00393                li      t2,11

100b8:       00730e33                add     t3,t1,t2

100bc:       05d00893                li      a7,93

100c0:       00000073                ecall
```

The option `-M no-aliases` allows us to see how the assembler *translated* the pseudo instruction `li` –

```
$ objdump -d -M no-aliases testprogram


testprogram:     file format elf64-littleriscv


Disassembly of section .text:

00000000000100b0 <_start>:

100b0:       00800313                addi    t1,zero,8

100b4:       00b00393                addi    t2,zero,11

100b8:       00730e33                add     t3,t1,t2

100bc:       05d00893                addi    a7,zero,93

100c0:       00000073                ecall
```

Since the immediate data was small (less than one byte) `li` was achieved using the single instruction `addi`.

2.3.7. Make

The commands that have been used so far for assembling and linking (`as,ld`) have worked well enough for our situation, however when multiple files are involved it is normal to use a build tool to accomplish this. The *make* utility keeps track of what has been done and will only apply actions to the changed portions. The instructions are conveyed to the utility using a *makefile*. The `makefile` below can be used to assemble link the program `testprogram.s`

Simple `makefile`

```
testprogram: testprogram.o

ld -o testprogram testprogram.o

testprogram.o: testprogram.s

as -o testprogram.o testprogram.s
```

The line at the top denotes the *target* file which *depends* on the *object* file which in turn is dependent on the *source* file. The rules on how to create the target file are shown above, so the flow is →

- Create the target file (`testprogram`) from the object file (`testprogram.o`) which is created from the source file (`testprogram.s`). The first target (here `testprogram`) is termed the *default goal*.

The make file is invoked by the command

```
make testprogram

as -o testprogram.o testprogram.s

ld -o testprogram testprogram.o
```

or in this case simply enter -

```
make

make: 'testprogram' is up to date.
```

Note use Tab characters for indentation in the `makefile`.

The next example assembles and links two programs into a single executable file.

```
OBJECTS = program1.o program2.o

all: myprogram

%.o : %.s

as $< -g -o $@

myprogram: $(OBJECTS)

ld -o myprogram $(OBJECTS)
```

This example will allow the target to be passed to the `makefile`:-

```
TARGETFILE = $(targetfile)

print: $(TARGETFILE).o

ld -o $(TARGETFILE) $(TARGETFILE).o

$(TARGETFILE).o: $(TARGETFILE).s

as -o $(TARGETFILE).o $(TARGETFILE).s
```

```
$ make targetfile=print

make: 'print' is up to date.
```

```
$ ls

makefile  print  print.o  print.s
```

2.4. Choosing a candidate platform

2.4.1. Hardware Platforms

Low-cost RISC-V hardware is available today, some hardware platforms that seem to work well are:

- VisionFive2 RISC-V Single Board Computer, StarFive JH7110 Processor with Integrated 3D GPU, 8GB Memory
  - starfivetech.com/en
- LicheePi 4A 64bit LPDDR4X 16GB RISC-V Single Board Computer.
  - https://wiki.sipeed.com/hardware/en/lichee/th1520/lp4a.html
- BananaPi BPI-F3
  - https://docs.banana-pi.org/en/BPI-F3/BananaPi_BPI-F3

2.4.2. Emulation and Simulation

An alternative is to use RISC-V emulation courtesy of QEMU which is an open-source emulator and virtualizer. See https://www.qemu.org/docs/master/ for more information.

Installation is covered in the next section.

Cross compilation is another option which is covered later.

2.4.2.1. Configuring a QEMU based Virtual machine

Note if using physical hardware the following steps can be skipped (if desired).

The following steps were performed to configure a Fedora based VM running on a Linux system using Fedora Workstation 42. Different software revisions may have to be tweaked accordingly. The installation steps are documented at The Fedora Project WIKI at Architectures/RISC-V/QEMU - Fedora Project Wiki

2.4.2.1.1. Install Qemu

```
sudo dnf install \
  libvirt-daemon-driver-qemu \
  libvirt-daemon-driver-storage-core \
  libvirt-daemon-driver-network \
  libvirt-daemon-config-network \
  libvirt-client \
  virt-install \
  qemu-system-riscv-core \
```

```
    edk2-riscv64
[sudo] password for fedorauser:
Updating and loading repositories:
Repositories loaded.
gpg: directory '/root/.gnupg' created
gpg: /root/.gnupg/trustdb.gpg: trustdb created
Package   "libvirt-daemon-driver-qemu-11.0.0-2.fc42.x86_64"   is   already
installed.
Package    "libvirt-daemon-driver-storage-core-11.0.0-2.fc42.x86_64"    is
already installed.
Package  "libvirt-daemon-driver-network-11.0.0-2.fc42.x86_64"  is  already
installed.
Package  "libvirt-daemon-config-network-11.0.0-2.fc42.x86_64"  is  already
installed.
Package "libvirt-client-11.0.0-2.fc42.x86_64" is already installed.
Package                                                Arch          Version
Repository                    Size
Installing:
 edk2-riscv64                                    noarch      20250221-8.fc42
fedora                    17.6 MiB
 qemu-system-riscv-core                        x86_64      2:9.2.3-1.fc42
. . .
```

2.4.2.1.2. Set up access and default URI

```
$ sudo usermod -a -G libvirt $(whoami)
= "qemu:///system"' >~/.config/libvirt/libvirt.conf
[fedorauser@fedora ~]$ sudo reboot
```

2.4.2.1.3. Get the image

```
[fedorauser@fedora  ~]$  wget  https://dl.fedoraproject.org/pub/alt/risc-
v/release/42/Cloud/riscv64/images/Fedora-Cloud-Base-Generic-42.20250414-
8635a3a5bfcd.riscv64.qcow2
Saving 'Fedora-Cloud-Base-Generic-42.20250414-8635a3a5bfcd.riscv64.qcow2'
. . .
```

2.4.2.1.4. Re-locate the image

```
$ sudo mv Fedora-Cloud-Base-Generic-42.20250414-8635a3a5bfcd.riscv64.qcow2
 /var/lib/libvirt/images/fedora-riscv.qcow2
```

2.4.2.1.5. Set up the environment and yml file

```
$ mkdir ~/riscv
cd riscv
vi user-data.yaml
====================================
#cloud-config
password: linux
chpasswd:
  expire: false


runcmd:
  - touch /etc/cloud/cloud-init.disabled
====================================
```

2.4.2.1.6. Set up the VM

Configure parameters, editing as required, such as RAM and CPU parameters

```
$ virt-install \
    --import \
    --name fedora-riscv \
```

```
    --osinfo fedora-rawhide \
    --arch riscv64 \
    --cpu mode=maximum \
    --vcpus 4 \
    --ram 8192 \
    --boot uefi \
    --disk path=/var/lib/libvirt/images/fedora-riscv.qcow2 \
    --network default \
    --tpm none \
    --graphics none \
    --controller scsi,model=virtio-scsi \
    --cloud-init user-data=user-data.yaml
. . .
Fedora Linux 42 (Cloud Edition)
Kernel 6.13.0-0.rc4.36.0.riscv64.fc42.riscv64 on riscv64 (ttyS0)
enp1s0: 192.168.122.132 fe80::5054:ff:fe43:927a
localhost login: [ 108.371505] cloud-init[982]: Cloud-init v. 24.2 running
'modules:final' at Thu, 15 May 2025 16:23:09 +0000. Up 107.79 seconds.
ci-info: no authorized SSH keys fingerprints found for user fedora.
<14>May               15               16:23:11               cloud-init:
#############################################################
<14>May 15 16:23:11 cloud-init: -----BEGIN SSH HOST KEY FINGERPRINTS-----
<14>May          15          16:23:11          cloud-init:          256
SHA256:WerTtl94f5Use//HZikpuOTZyJzztfVWhGBhP0McjZU root@localhost (ECDSA)
```

```
<14>May          15          16:23:11          cloud-init:          256
SHA256:FEOBA1tWS12IOewDzRywk0HOjkqZ2x66Rx++4LHkwO8          root@localhost
(ED25519)
. . .
[ 109.684931] cloud-init[982]: Cloud-init v. 24.2 finished at Thu, 15 May
2025       16:23:11       +0000.       Datasource       DataSourceNoCloud
[seed=/dev/sr0][dsmode=net]. Up 109.56 seconds
Log on wuh username “fedora” and password “linux”
localhost login: fedora
Password:
```

2.4.2.1.7. Check the architecture -

```
[fedora@localhost ~]$ lscpu
Architecture: riscv64
Byte Order: Little Endian
CPU(s): 4
On-line CPU(s) list: 0-3
Vendor ID: 0x0
Model name: -
CPU family: 0x0
Model: 0x0
Thread(s) per core: 1
Core(s) per socket: 4
. . .
```

2.4.2.1.8. Check RAM size

```
$ free -m
total used free shared buff/cache available
```

```
Mem: 7911 348 7549 0 164 7562

Swap: 7910 0 7910
```

2.4.2.1.9. Test the coding environment

Install tools

```
$ sudo dnf install -y binutils

Updating and loading repositories:

Fedora RISC-V 42 100% | 1.1 MiB/s | 17.3 MiB | 00m16s

Fedora RISC-V 42 - Staging 100% | 291.8 KiB/s | 453.4 KiB | 00m02s

Repositories loaded.

Package Arch Version Repository Size

Installing:

binutils riscv64 2.44-3.fc42 fedora-riscv 24.3 MiB

Transaction Summary:

Installing: 1 package

. . .
```

Create a small assembly file

```
vi test.s

.global _start

_start:

la a1, hellorisc

addi a2, x0, 14

addi a7, x0, 64

ecall

addi a0, x0, 0

addi a7, x0, 93

ecall
```

```
.data

hellorisc:.ascii "Hello RISC-V!\n"
```

Assemble, link and execute

```
$ as -g -o test.o test.s

[fedora@localhost ~]$ ls

test.o test.s

[fedora@localhost ~]$ ld -o test test.o

[fedora@localhost ~]$ chmod 777 test

[fedora@localhost ~]$ ./test

Hello RISC-V!
```

2.4.2.1.10. Shutdown and restarting

Shutdown the machine with `$ sudo poweroff`

Restart with `$ virsh start fedora-riscv –console`

2.4.2.1.11. Optional activities

Add network tools

```
$ sudo dnf install -y net-tools
```

2.4.2.1.12. Grow the virtual disk capacity

The virtual disk image can be expanded with the command -

```
qemu-img resize /var/lib/libvirt/images/fedora-riscv.qcow2 +20G
```

This adds 20G capacity to the existing image. The next task is to boot the image and grow a partition (here partition3 will be expanded) with the `fdisk` utility provided by the virtual machine

```
[fedora@localhost ~]$ sudo fdisk /dev/vda

Welcome to fdisk (util-linux 2.40.4).

Changes will remain in memory only, until you decide to write them.

Be careful before using the write command.

. . .

Command (m for help): p
```

```
Disk /dev/vda: 25 GiB, 26843545600 bytes, 52428800 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: D8EE907C-0F17-41BA-BBEC-8A0DA4FB0950
Device Start End Sectors Size Type
/dev/vda1 2048 206847 204800 100M EFI System
/dev/vda2 206848 2254847 2048000 1000M Linux extended boot
/dev/vda3 2254848 10485726 8230879 3.9G Linux root (RISC-V-64)
Command (m for help): e
Partition number (1-3, default 3):
New <size>{K,M,G,T,P} in bytes or <size>S in sectors (default 23.9G):
Partition 3 has been resized.
Command (m for help): p
Disk /dev/vda: 25 GiB, 26843545600 bytes, 52428800 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: D8EE907C-0F17-41BA-BBEC-8A0DA4FB0950
Device Start End Sectors Size Type
/dev/vda1 2048 206847 204800 100M EFI System
/dev/vda2 206848 2254847 2048000 1000M Linux extended boot
/dev/vda3 2254848 52428766 50173919 23.9G Linux root (RISC-V-64)
Command (m for help): w
```

```
The partition table has been altered.

Syncing disks.
```

The `fdisk` utility has expanded the partition to 24G compared to the previous capacity of 4GB.

Verify from the Operating System -

```
[fedora@localhost ~]$ lsblk

NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINTS

sr0 11:0 1 1024M 0 rom

zram0 251:0 0 7.7G 0 disk [SWAP]

vda 252:0 0 25G 0 disk

├─vda1 252:1 0 100M 0 part /boot/efi

├─vda2 252:2 0 1000M 0 part /boot

└─vda3 252:3 0 23.9G 0 part /var

/home

/
```

2.4.2.2. Updating Fedora

```
$ sudo dnf upgrade --best

Fedora RISC-V                          4.1 kB/s | 3.8 kB     00:00

Dependencies resolved.

================================================================================

Package                Arch    Version                   Repository    Size

================================================================================

Installing:

iwlegacy-firmware    noarch  20230804-153.fc38       fedora-riscv 140

. . .
```

2.4.2.3. Simulators

Simulators are as the name suggests program that run on the native system, providing the functionality of a target system. They can normally be run online or perhaps under the control of an environment such as Java. Two such programs are:

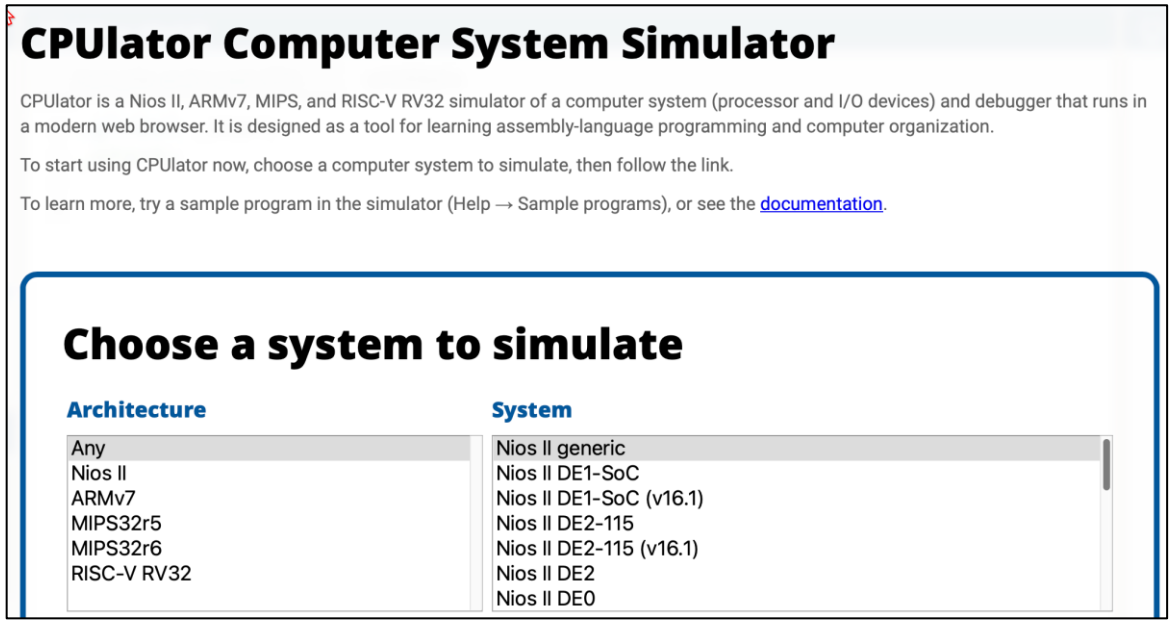
- CPUlator
- RARS

2.4.2.3.1. CPUlator

CPUlator[30] is an online simulator that supports RV32. The simulator is an excellent tool for debugging as it provides *single stepping* through the code, as well as showing memory, registers and code disassembly.

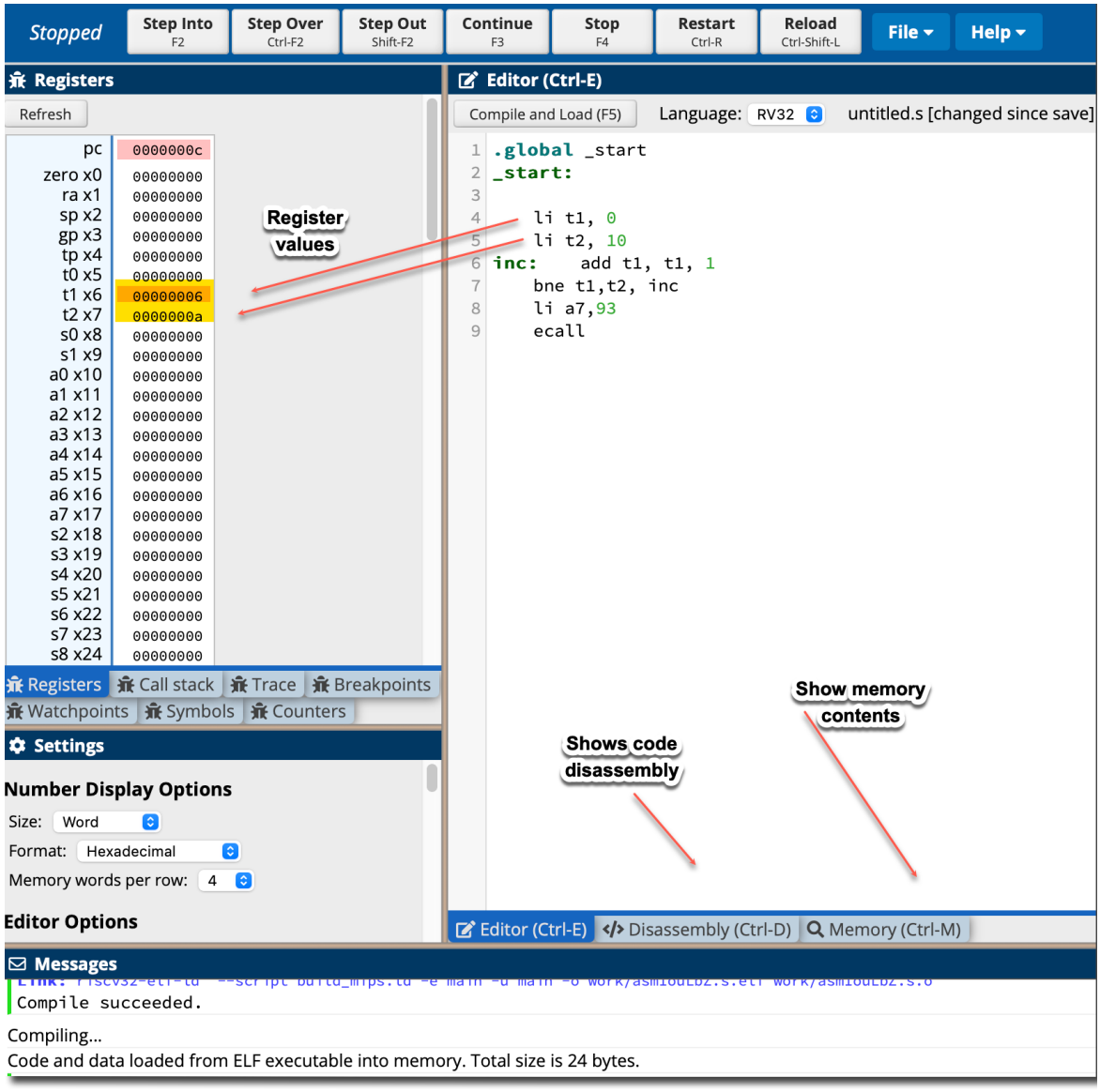
To get started select the system to be emulated (here RISC-V RV32) as shown in Figure 2-2 and enter code. The code can be made executable in the simulator by selecting <Compile and Load> as shown in Figure 2-3. The code can be executed one line at a time by selecting <Step Into>. Each step will show changes to the register and memory contents.

Figure 2-2 CPUlator home page


CPUlator Computer System Simulator
CPUlator is a Nios II, ARMv7, MIPS, and RISC-V RV32 simulator of a computer system (processor and I/O devices) and debugger that runs in a modern web browser. It is designed as a tool for learning assembly-language programming and computer organization.
To start using CPUlator now, choose a computer system to simulate, then follow the link.
To learn more, try a sample program in the simulator (Help → Sample programs), or see the documentation.
Choose a system to simulate
Architecture
Any
Nios II
ARMv7
MIPS32r5
MIPS32r6
RISC-V RV32
System
Nios II generic
Nios II DE1-SoC
Nios II DE1-SoC (v16.1)
Nios II DE2-115
Nios II DE2-115 (v16.1)
Nios II DE2
Nios II DE0

[30] The URL for CPUlator is https://cpulator.01xz.net/

Figure 2-3 Compiling and executing code with CPUlator

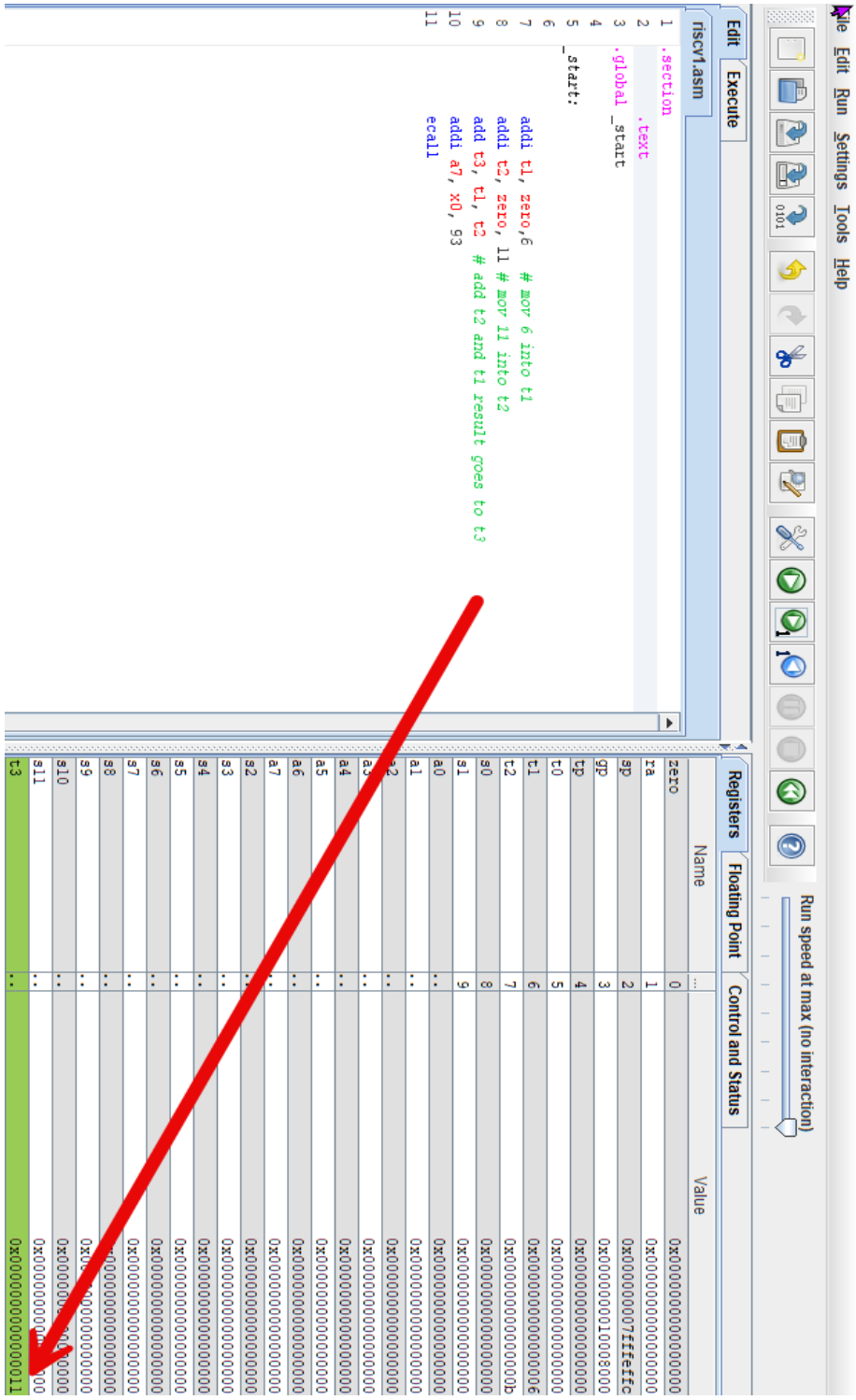


2.4.2.3.2. RARS

RARS[31] stands for RISC-V Assembler and Runtime Simulator. It is also an excellent tool for learning RISC-V assembly language.

[31] The URL for RARS is https://github.com/TheThirdOne/rars/releases/tag/continuous

Figure 2-4 RARS Execution screen



Refer to the URL in the footnotes for more information on CPUlator and RARS.

In this example the downloaded RARS version was `rars_3897cfa.jar` as shown in Figure 2-5. To run execute `java -jar rars_3897cfa.jar`.

Figure 2-5 Downloading RARS

The following link lists more simulators – https://www.riscvschool.com/risc-v-simulators/

2.4.3. Using strace

The *strace* utility can be used to monitor which syscalls have been invoked by a particular program or process: -

```
$ strace -c ./print

Hello again!

% time     seconds  usecs/call     calls    errors syscall

------ ----------- ----------- --------- --------- ----------------

0.00    0.000000           0         1           write

0.00    0.000000           0         1           execve

------ ----------- ----------- --------- --------- ----------------

100.00    0.000000           0         2           total
```

`Strace,` here shows that the syscalls `write` and `exit` were invoked once.

RISC-V Instructions Covered in Chapter 2

1. **addi** Add Immediate Example: addi t2, zero, 11
2. **add** Add (register-to-register) Example: add t3, t1, t2
3. **ecall** – Environment call (used for syscalls)
4. **lui** – Load Upper Immediate
5. **auipc** – Add Upper Immediate to PC
6. **sw** – Store Word
7. **jal / jalr** – Jump and Link / Jump and Link Register (implicitly discussed in context of control transfer)

Exercises for chapter 2

1. What qualifier would you add to the `as` command to embed debug information?
2. What is the purpose of a linker?
3. How many registers are available for general purpose use?
4. What are assembly directives?
5. What are syscalls?
6. What is the function of a makefile?
7. What are assembly aliases?
8. What tool is used to disassemble an executable program

Chapter 3. Dealing with memory

Overview of the chapter

Chapter 3 focuses on how RISC-V assembly interacts with memory, introducing key concepts such as loading, storing, and addressing. It builds on previous chapters by explaining how data is accessed, moved, and manipulated in memory during program execution.

3.1. Load and Store instructions

Memory addresses are *loaded* from memory into registers, and *stored* back from registers to memory. Operations are with respect to memory so loading from memory to registers is a *read* operation and storing from registers is a *write* operation. The method by which memory addresses are derived is known as addressing modes and there are several. The code fragments in this chapter will show how to communicate with memory and will also introduce various addressing modes.

Load and store instructions can access memory. Data is loaded from memory, acted on and then stored back to memory. This is termed *load-store architecture*.

3.1.1. LOAD Instructions (Memory → Registers)

3.1.1.1. Examining memory with GDB

GDB can be used to examine memory. The format of the command is `x/nfu addr`. Here the parameters have the following meaning:

Table 3-1 Using GDB to display memory contents

n	**How much memory to display in units, with a default value of one.**
f	This is the display format; default is to display in hex. The main options are o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char), s(string)
u	Unit size b = byte h = halfword (2 bytes) w = word (4 bytes) g = giant (8 bytes)

Example

```
(gdb) x/16w  0x4100e0
0x4100e0:    0x6c6c6548  0x00000a6f  0x00000000  0x00000000
0x4100f0:    0x0000002c  0x00000002  0x00080000  0x00000000
0x410100:    0x004000b0  0x00000000  0x00000028  0x00000000
0x410110:    0x00000000  0x00000000  0x00000000  0x00000000
```

To examine memory pointed to by a label (`mymemorylocation`) the following syntax can be used –

```
(gdb) x /16xw &mymemorylocation
0x11104:    0x0000abcd  0x00001234  0x00000000  0x00000000
0x11114:    0x36410000  0x72000000  0x76637369  0x002c0100
0x11124:    0x72050000  0x69343676  0x5f307032  0x3070326d
0x11134:    0x7032615f  0x32665f30  0x645f3070  0x5f307032
```

3.1.1.2. Load and Store example

Listing 3-1 below shows a basic example of how to read from and write to memory.

The first instruction `lw, t0, word1` loads a word of data into register t0. The data is identified by the label `word1` in the `.data` section of the code.

The next instruction `la t1, bufferspace` is the destination address for the data that will be loaded into the register t1. The address is identified by the label `bufferspace.`

The third instruction `sw t0, 0(t1)` stores the word held in t0 into the memory address pointed to by register t1.

Listing 3-1 Basic read (load) and write (store) memory operation

```
.section .text

.global _start

_start:

lw      t0, word1 #Reads in the value 0xabcd into register t0

la    t1, bufferspace # Load the address pointed to by bufferspace into t1

sw      t0, 0(t1)    # Store the value of the word held in register t0 into

the memory location pointed to by register t1 plus an offset of 0.

addi a7, x0, 93

ecall

.data

word1: .4byte 0xabcd, 0x1234

bufferspace: .space 10
```

A trace with GDB is instructive, note the comments (not part of the GDB output) are shown in italicized font)

```
gdb listing3-1

. . .

Reading symbols from listing3-1...
```

Next set a breakpoint

```
(gdb) b 1

Breakpoint 1 at 0x100e8: file listing3-1.s, line 5.
```

Show the memory contents at the location word1

```
(gdb) x /4xw &word1

0x11104:    0x0000abcd  0x00001234  0x00000000  0x00000000
```

Show that the data at location bufferspace is uninitialized

```
(gdb) x /2xw &bufferspace

0x1110c:    0x00000000  0x00000000
```

Start the program

```
(gdb) run

Starting program: /home/alan/asm/chapter03/listing3-1

Breakpoint 1, _start () at listing3-1.s:5

5 lw    t0, word1 #Reads in the value 0xabcd into register t0

(gdb) n

6           la    t1, bufferspace

(gdb) n

7           sw      t0, 0(t1)     # Store the value of the word  . . .
```

Show that register t0 now holds the word 0xabcd

```
(gdb) i r t0

t0             0xabcd   43981
```

Show that register t1 now holds the address pointed to by bufferspace, note that this was shown to be 0x1110c earlier in this trace.

```
(gdb) i r t1

t1             0x1110c  69900
```

Show that location bufferspace now holds the value 0xabcd

```
(gdb) x /2xw &bufferspace

0x1110c:    0x0000abcd  0x00000000
```

3.2. Outputting (Writing) ASCII text

The next listing shows how text can be sent to the standard output device (stdout) – the screen. The *write* syscall will do this job and it has the decimal value of 64. Three registers (a0, a1, a2) hold the parameters that are required by this syscall and are set up as shown in Table 3-2.

Table 3-2 Parameters required by the Write syscall

Register	Parameter meaning
a0	Holds value 1 (stdout)
a1	Hold the address of the output text (located at the label message)
a2	Contains the length of the output text (12 characters)

Note the message string is terminated by the *newline* character (/n)

Listing 3-2 Use of the Write Syscall

```
# listing3-2.s

.section .text

.global _start

_start:

li a0, 1 # use a0 for stdout

la a1, message # Load the address of the message text

li a2, 12 # Store the message length

li a7, 64   # Write syscall

ecall

li a7, 93  # Exit syscall

ecall

.data

message: .ascii "Hello RISCV\n"
```

Execute the program with the command –

```
./listing3-2

Hello RISCV
```

The unaliased version of this program is shown below:

```
objdump -d -M no-aliases listing3-2

listing3-2:     file format elf64-littleriscv

Disassembly of section .text:

00000000000100e8 <_start>:

100e8:       00100513           addi    a0,zero,1

100ec:       00001597           auipc   a1,0x1

100f0:       01c58593           addi    a1,a1,28 # 11108 <__DATA_BEGIN__>

100f4:       00c00613           addi    a2,zero,12

100f8:       04000893           addi    a7,zero,64

100fc:       00000073           ecall

10100:       05d00893           addi    a7,zero,93

10104:       00000073           ecall
```

GDB can be used to show the memory layout -

```
(gdb) x /16c &message

0x11108:   72 'H'  101 'e' 108 'l' 108 'l' 111 'o' 32 ' '  82 'R'  73 'I'

0x11110:  83 'S'  67 'C'  86 'V'  10 '\n' 65 'A'  54 '6'  0 '\000' 0 '\000'
```

This shows that the ASCII characters are laid out starting at the lowest address (0x11108) then counting upwards to 0x 11113.

3.3. Inputting (reading) values

The next example shows how to read in a value using the read syscall. The read syscall uses the value 63 and places the input into memory defined by the symbol `buffer.`

Table 3-3 Parameters required by the read syscall

Register	Parameter meaning
a0	Holds the value 1 (stdin)
a1	Hold the address of the storage buffer
a2	Contains the length of the input characters

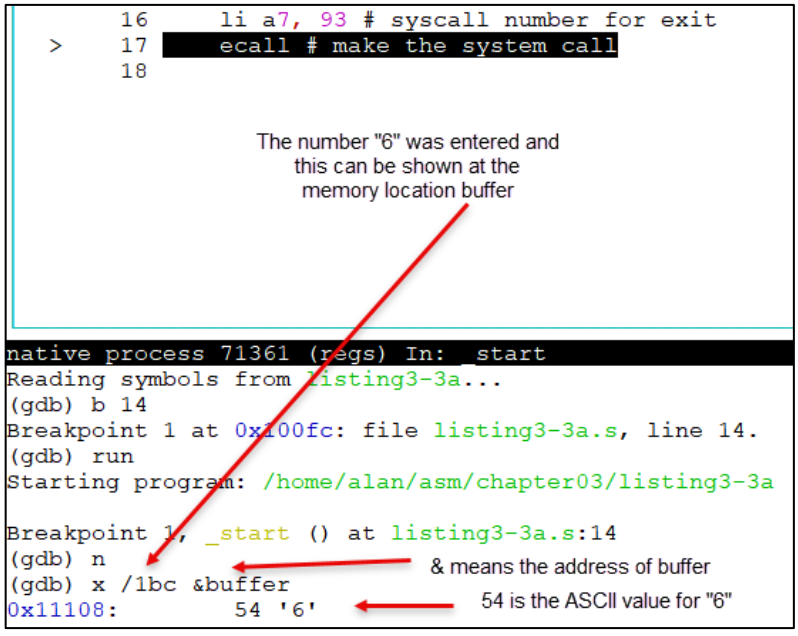
Listing 3-3 Input operation

 # This is a simple program that reads in a single digit

```
.section .data
```

```
buffer:

.space 1

.section .text

.global _start

_start:

li a0, 0 # file descriptor 0 (stdin)

la a1, buffer # address of the buffer

li a2, 1 # number of bytes to read

li a7, 63 # Read syscall

ecall

li a7, 93 # syscall number for exit

ecall # make the system call
```

The GDB session below shows that the memory location buffer holds the value 54 decimal which is the ASCI code of the character “6”.


16 li a7, 93 # syscall number for exit
> 17 ecall # make the system call
18
The number "6" was entered and this can be shown at the memory location buffer
native process 71361 (regs) In: _start
Reading symbols from listing3-3a...
(gdb) b 14
Breakpoint 1 at 0x100fc: file listing3-3a.s, line 14.
(gdb) run
Starting program: /home/alan/asm/chapter03/listing3-3a
Breakpoint 1, _start () at listing3-3a.s:14
(gdb) n
(gdb) x /1bc &buffer
0x11108: 54 '6'
& means the address of buffer
54 is the ASCII value for "6"

3.4. Relative and absolute addressing

Program Counter (PC) relative addressing is used to reference locations relative to the program counter. For example, a location could be accessed as PC +100 which would refer to a location 100 places beyond the current program counter's contents. Execution of consecutive(non-branch/jump) instructions advances the program counter by four, since instructions have a width of 32-bits (4 bytes). It is important to facilitate forward and backward locations. Absolute addressing refers to the actual location in memory where an instruction or data resides.

3.4.1. RISC-V Assembler Modifiers

The assembler supports instructions to generate relative and absolute addresses. The address is broken up into the 12-bit lower portion (lo) and a 20-bit upper portion (hi). Before discussing this topic in detail - consider how the pseudo instruction `la` breaks into the instructions `auipc` and `addi`:

```
la a1, message
```

Disassembles to →

```
00000000000100e8 <_start>:

100e8:       00100513                li      a0,1

100ec:       00001597                auipc   a1,0x1

100f0:      01c58593             addi    a1,a1,28 # 11108 <__DATA_BEGIN__>
```

The upper 20 bits (from `auipc`) are taken from the Program Counter's current contents (0x110ec) and the lower 12 bits (from `addi`) are 0x01c.

The immediate value of 0x1 is placed in the a1 register (from the `auipc` instruction) and is then shifted 12 places, causing it to occupy the upper 20 bits of the register. Register a1 now holds the value 0X00001000. This is added to the value in the Program Counter giving 0x110ec. Next the immediate value (0x1C) (from the `addi` instruction) is added to the contents of a1 and placed in register a1, so a1 now contains 0x11108.

The steps are shown below in Figure 3-1.

Figure 3-1 AUIPC and ADDI instruction example to generate an address

Using the GDB command `info variables` shows:

```
All defined variables:

Non-debugging symbols:

0x0000000000011108  __DATA_BEGIN__

0x0000000000011108  message

0x000000000001111f  __SDATA_BEGIN__

0x000000000001111f  __bss_start

0x000000000001111f  _edata

0x0000000000011120  __BSS_END__

0x0000000000011120  _end
```

This confirms that the address of the string message resides at 0x11108. It also shows the value of the pseudo instruction `la` which is much easier to use. The assembler also helps us if we do not use the pseudo instructions. The instructions can resolve addresses by using modifiers such as `%lo, %hi, %pcrel_hi` and `%pcrel_lo`.

Table 3-4 Absolute and relative adressing

Modifier	Format/Example	Description
%hi	`lui a1, %hi(symbol)`	Loads upper 20 bits of the symbol's address into register a1
%lo	`addi a1, a1, %lo(symbol`	Loads lower 12 bits of the symbol's address into register a1
%pcrel_hi	`auipc a2, %pcrel_hi(symbol)`	Loads the high 20 bits of a relative address between the PC and symbol
%pcrel_lo	`addi a2, a2, %pcrel_lo(label)`	Loads the high 20 bits of a relative address between the PC and label

The reason that two instructions are needed is that there is no single instruction that is capable of loading a 32-bit immediate value. Referring back to the I-type and U-type instructions on page 2-7, there are instructions that load 12 bits and instructions that load 20 bits. Combining them is how a 32-bit immediate value is achieved.

- LUI is a U-type instruction and sets the low order bits to zero in the destination register and fills in the high order bits.
- ADDI is an I-type instruction and adds in the low order bits to the destination register.
- AUIPC sets the destination register's high order bits to the sum of the immediate value and the program counter with the lo order bits set to zero.

The next listing shows an example of PC-Relative addressing.

Listing 3-4 PC-Relative addressing example

```
/* Listing 3-4

This listing shows how to use PC-Relative addressing

using modifiers*/


.section .data

message:

.ascii "This is a line of text\n"

.equ writecall, 64

.equ exitcall, 93

.equ stout, 1

.equ stringlength, 23


.section .text

.global _start

_start:

li a0, stout # stdout

label1: auipc a1,%pcrel_hi(message) # Loads upper 20 bits

addi a1,a1,%pcrel_lo(label1) # Loads lower 12 bits

li a2, stringlength # String length

li a7, writecall # Write syscall

ecall


li a7, exitcall # syscall number for exit

ecall # make the system call
```

Listing 3-5 shows how absolute addressing is achieved with `%lo` and `%hi`.

Listing 3-5 Using absolute addressing with %lo and %hi

```
This listing shows how to generate absolute addressing

using %lo and %hi modifiers*/

.section .data

message:

.ascii "This is a line of text\n"

.equ writecall, 64

.equ exitcall, 93

.equ stout, 1

.equ stringlength, 23


.section .text

.global _start

_start:

li a0, stout # stdout

lui a1, %hi(message) # Loads upper 20 bits of mesages' absolute address

addi a1,a1,%lo(message) # Loads lower 12 bits of message's absolute address

li a2, stringlength # String length

li a7, writecall # Write syscall

ecall

li a7, exitcall # syscall number for exit

ecall # make the system call
```

The first absolute addressing instruction `lui ai, %hi(message)` loads a1 with the value 0x11000, the next instruction `addi a1, a1, %lo(message)` adds in 0x108 to a1 giving a final result of 0x11108 which is the absolute address of the symbol *message*.

3.5. Linker Relaxation

The directive `.option norelax` is used to disable *linker relaxation*[32]. Relaxation is used to optimize performance by reducing the number of instructions when the address range is limited. This is illustrated in the next two listings.

Listing 3-6 Non relaxed version of code

```
# This version does not use relaxation

.section .data

ask:

.ascii "Please input a character\n"

.align 2

confirm:

.ascii "You entered: \n "

.align 2

linefeed:

.ascii "\n"

buffer:

.space 4


.section .text

.global _start

_start:

.option push # Save context

.option norelax # Turn off relaxation to set up the global pointer

1:auipc gp, %pcrel_hi(__global_pointer$)
```

[32] See https://www.sifive.com/blog/all-aboard-part-3-linker-relaxation-in-riscv-toolchain for more information on relaxation.

```
addi  gp, gp, %pcrel_lo(1b)# b for back
.option pop # Now restore relaxation
.option norelax
li a0, 1 #stdout
la a1, ask #Text for first output string
li a2, 27 #String length
li a7, 64 # Write syscall
ecall
li a0, 0 # file descriptor 0 (stdin)
la a1, buffer # address of the buffer
li a2, 1 # number of bytes to read
li a7, 63 # Read syscall
ecall


li a0, 1 #stdout again
la a1, confirm # Text for second output string
li a2, 15#Length
li a7, 64 #Write syscall
ecall
li a0, 1 #stdout again
la a1, buffer
li a2, 1 #Length
li a7, 64 #Write syscall
ecall


# Tidy up with a newline!
```

```
li a0, 1

la a1, linefeed

li a2, 1

li a7, 64

ecall

li a7, 93 # syscall number for exit

ecall # make the system call
```

The numeric label 1 is suffixed with ‘b’ or ‘f’ for backward and forward references respectively.

Listing 3-7 Relaxed version of code

```
# This version uses relaxation (Default mode)

.section .data

ask:

.ascii "Please input a character\n"

.align 2

confirm:

.ascii "You entered: \n "

.align 2

linefeed:

.ascii "\n"

buffer:

.space 4

.section .text

.global _start

_start:

.option push # Save context

.option norelax # Turn off relaxation to set up the global pointer
```

```
1: auipc gp, %pcrel_hi(__global_pointer$)
addi  gp, gp, %pcrel_lo(1b)
.option pop # Now restore relaxation state
# .option norelax is commented out in this version
li a0, 1 #stdout
la a1, ask #Text for first output string
li a2, 27 #String length
li a7, 64 # Write syscall
ecall

li a0, 0 # file descriptor 0 (stdin)
la a1, buffer # address of the buffer
li a2, 1 # number of bytes to read
li a7, 63 # Read syscall
ecall

li a0, 1 #stdout again
la a1, confirm # Text for second output string
li a2, 15#Length
li a7, 64 #Write syscall
ecall
li a0, 1 #stdout again
la a1, buffer
li a2, 1 #Length
li a7, 64 #Write syscall
ecall
```

```
# Tidy up with a newline!
li a0, 1
la a1, linefeed
li a2, 1
li a7, 64
ecall
li a7, 93 # syscall number for exit
ecall # make the system call
```

Register X3, (GP, the global pointer register) is loaded with a value that fits into the 12-bit address space of the .data section. The directives `.option push` and `.option pop` restores the relaxation option to its previous state. The default condition of relaxation is *enabled*.

Table 3-5[33] shows how relaxation reduces the code size and enhances performance. Since the contents of the .data section are small enough to fit into 12 bits, the upper 20 bits need not be fetched each time.

The real gain is not so much the size of the code but with performance. Code that uses repetitive loop iterations can benefit greatly in terms of reduction of execution time.[34]

Table 3-5 Comparison of relaxed and non-relaxed code

Listing 3-6 Non relaxed version of code	**Listing 3-7 Relaxed version of code**
00000000000100e8 <_start>:	**00000000000100e8 <_start>:**
100e8: 00002197 auipc gp,0x2	**100e8: 00002197 auipc gp,0x2**
100ec: 88818193 addi gp,gp,-1912 # 11970 <__global_pointer$>	**100ec: 87818193 addi gp,gp,-1928 # 11960 <__global_pointer$>**
100f0: 00100513 addi a0,zero,1	**100f0: 00100513 addi a0,zero,1**
100f4: 00001597 auipc a1,0x1	**100f4: 00001597 auipc a1,0x1**
100f8: 07c58593 addi a1,a1,124 # 11170 <__DATA_BEGIN__>	**100f8: 06c58593 addi a1,a1,108 # 11160 <__DATA_BEGIN__>**
100fc: 01b00613 addi a2,zero,27	**100fc: 01b00613 addi a2,zero,27**
10100: 04000893 addi a7,zero,64	**10100: 04000893 addi a7,zero,64**

[33] The listings were generated by `objdump -d -M no-aliases <file>`

[34] Refer to RISC-V ABIs Specification (https://lists.riscv.org/g/tech-psabi/attachment/61/0/riscv-abi.pdf) section 8.5.5 for more information on the global offset table.

```
10104:  00000073        ecall
10108:  00000513        addi    a0,zero,0
1010c:  00001597        auipc   a1,0x1
10110:  09158593        addi    a1,a1,145 # 1119d <buffer>
10114:  00100613        addi    a2,zero,1
10118:  03f00893        addi    a7,zero,63
1011c:  00000073        ecall
10120:  00100513        addi    a0,zero,1
10124:  00001597        auipc   a1,0x1
10128:  06858593        addi    a1,a1,104 # 1118c <confirm>
1012c:  00f00613        addi    a2, zero,15
10130:  04000893        addi    a7, zero,64
10134:  00000073        ecall
10138:  00100513        addi    a0, zero,1
1013c:  00001597        auipc   a1,0x1
10140:  06158593        addi    a1,a1,97 # 1119d <buffer>
10144:  00100613        addi    a2,zero,1
10148:  04000893        addi    a7,zero,64
1014c:  00000073        ecall
10150:  00100513        addi    a0,zero,1
10154:  00001597        auipc   a1,0x1
10158:  04858593        addi    a1, a1,72 # 1119c <linefeed>
1015c:  00100613        addi    a2, zero,1
10160:  04000893        addi    a7, zero,64
10164:  00000073        ecall
10168:  05d00893        addi    a7, zero,93
1016c:  00000073        ecall
```

```
10104:  00000073        ecall
10108:  00000513        addi    a0,zero,0
1010c:  82d18593        addi    a1,gp,-2003 # 1118d <buffer>
10110:  00100613        addi    a2,zero,1
10114:  03f00893        addi    a7, zero,63
10118:  00000073        ecall
1011c:  00100513        addi    a0, zero,1
10120:  81c18593        addi    a1, gp,-2020 # 1117c <confirm>
10124:  00f00613        addi    a2, zero,15
10128:  04000893        addi    a7, zero,64
1012c:  00000073        ecall
10130:  00100513        addi    a0, zero,1
10134:  82d18593        addi    a1,gp,-2003 # 1118d <buffer>
10138:  00100613        addi    a2, zero,1
1013c:  04000893        addi    a7, zero,64
10140:  00000073        ecall
10144:  00100513        addi    a0, zero,1
10148:  82c18593        addi    a1,gp,-2004 # 1118c <linefeed>
1014c:  00100613        addi    a2, zero,1
10150:  04000893        addi    a7, zero,64
10154:  00000073        ecall
10158:  05d00893        addi    a7, zero,93
1015c:  00000073        ecall
```

Listing 3-6 auipc count

```
100e8:        00002197                auipc   gp,0x2
```

```
100f4:       00001597                auipc   a1,0x1

1010c:       00001597                auipc   a1,0x1

10124:       00001597                auipc   a1,0x1

1013c:       00001597                auipc   a1,0x1

10154:       00001597                auipc   a1,0x1
```

Listing 3-7 auipc count

```
100e8:       00002197                auipc   gp,0x2

100f4:       00001597                auipc   a1,0x1
```

The assembler can be modified to generate non-relaxed code with the `-mno-relax` option. To modify the make file to include it edit the makefile to read –

```
TARGETFILE = $(targetfile)

print: $(TARGETFILE).o

ld -o $(TARGETFILE) $(TARGETFILE).o

$(TARGETFILE).o: $(TARGETFILE).s

as -mno-relax -g -o $(TARGETFILE).o $(TARGETFILE).s
```

For the remaining programs in the book the `makefiles`[35] (unless stated otherwise) will include the `-mno-relax` option.

3.5.1. Enhancements to GDB

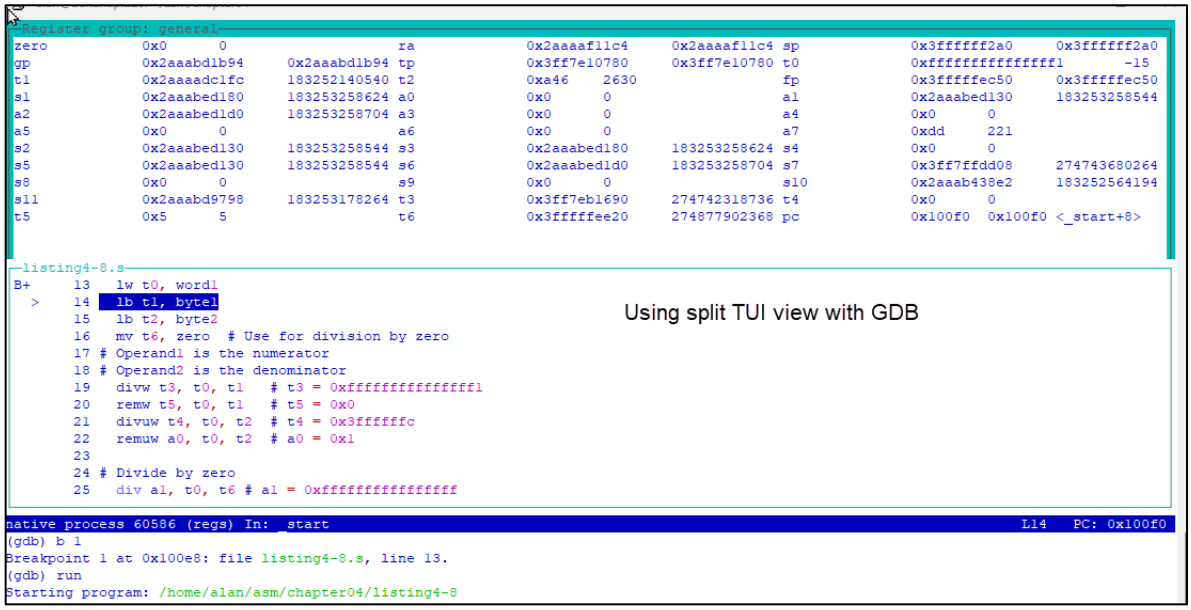
So far GDB has been used in default mode for analyzing code. Entering the following commands into the file `~/.gdbinit` will give a better (TUI) layout experience.

```
layout split

layout regs

set history save on

set history filename ~/gdbhistory

set logging enabled on
```

[35] For reasons already discussed linker relaxation can be helpful in performance-oriented applications and then a different makefile would be used.

Note that if using the GDB TUI then the up and down arrows are no longer available for command history; use Ctrl-P(revious) and Ctrl-N(ext) instead.

Figure 3-2 GDB using TUI


Register group: general
zero 0x0 0 ra 0x2aaaaf11c4 0x2aaaaf11c4 sp 0x3ffffff2a0 0x3ffffff2a0
gp 0x2aaabd1b94 0x2aaabd1b94 tp 0x3ff7e10780 0x3ff7e10780 t0 0xfffffffffffffff1 -15
t1 0x2aaaadc1fc 183252140540 t2 0xa46 2630 fp 0x3fffffec50 0x3fffffec50
s1 0x2aaabed180 183253258624 a0 0x0 0 a1 0x2aaabed130 183253258544
a2 0x2aaabed1d0 183253258704 a3 0x0 0 a4 0x0 0
a5 0x0 0 a6 0x0 0 a7 0xdd 221
s2 0x2aaabed130 183253258544 s3 0x2aaabed180 183253258624 s4 0x0 0
s5 0x2aaabed130 183253258544 s6 0x2aaabed1d0 183253258704 s7 0x3ff7ffdd08 274743680264
s8 0x0 0 s9 0x0 0 s10 0x2aaab438e2 183252564194
s11 0x2aaabd9798 183253178264 t3 0x3ff7eb1690 274742318736 t4 0x0 0
t5 0x5 5 t6 0x3fffffee20 274877902368 pc 0x100f0 0x100f0 <_start+8>
listing4-8.s
B+ 13 lw t0, word1
> 14 lb t1, byte1
15 lb t2, byte2
16 mv t6, zero # Use for division by zero
17 # Operand1 is the numerator
18 # Operand2 is the denominator
19 divw t3, t0, t1 # t3 = 0xffffffffffffffff
20 remw t5, t0, t1 # t5 = 0x0
21 divuw t4, t0, t2 # t4 = 0x3fffffc
22 remuw a0, t0, t2 # a0 = 0x1
23
24 # Divide by zero
25 div a1, t0, t6 # a1 = 0xffffffffffffffff
Using split TUI view with GDB
native process 60586 (regs) In: _start L14 PC: 0x100f0
(gdb) b 1
Breakpoint 1 at 0x100e8: file listing4-8.s, line 13.
(gdb) run
Starting program: /home/alan/asm/chapter04/listing4-8

Exercises for chapter3

1. Write a program that takes a user inputted string, printing out hexadecimal codes for each character in the string, for example –

"This is the input string"

character	Hex value
T	54
h	68

. . .

2. Describe the purpose of linker relaxation.

RISC-V instructions covered in chapter 3

Load Instructions:

lb – Load byte

lbu – Load byte unsigned

lh – Load halfword

lhu – Load halfword unsigned

lw – Load word

lwu – Load word unsigned (64-bit systems)

ld – Load doubleword

Store Instructions:

sb – Store byte

sh – Store halfword

sw – Store word

sd – Store doubleword

System Call and Immediate Instructions:

li – Load immediate (pseudo-instruction)

la – Load address (pseudo-instruction)

ecall – Environment call (used for invoking syscalls)

Addressing & Assembler-Related:

auipc – Add upper immediate to PC (used in PC-relative addressing)

Assembler modifiers like %lo(symbol) and %hi(symbol) are also discussed to support **absolute addressing**.

Chapter 4. Arithmetic operations (First Pass)

Overview of the chapter

Chapter 4 focuses on arithmetic and logical operations in RISC-V assembly. It builds upon the memory-handling concepts of Chapter 3 by introducing how to perform calculations, data manipulation, and conditional logic using registers. Floating-point operations are deferred until page 8-1.

4.1. Data Sizes

RISC_V uses the data sizes listed in Table 4-1.

Table 4-1 Data Types

# of bits	Definition
8	Byte
16	Halfword
32	Word
64	Doubleword

Load and Store instructions designate variants of these data sizes with the following abbreviations:

- W: Word
- H: Halfword
- HU: Halfword unsigned
- B: Byte
- BU: Byte unsigned

4.2. Integer Instructions

4.2.1. Register ADD

The first listing shows the ADD instruction which has the format `add rd`(estination), `rs`(ource)`1`, `rs`(ource)`2`. In this case the addition of source register t0 and source register t1 are sent to the destination register t3. The ADDW

Listing 4-1 Basic Addition using ADD

```
/* Listing 4-1 Simple addition (register)instruction

32-bit (add) and 64-bit (addw) are shown */
```

```
.section .data

.equ wordnumber1, 0xffdc5678

.equ wordnumber2, 0x4321

.section .text

.global _start

_start:

li t0, wordnumber1

li t1, wordnumber2

add t3, t0,t1     #32-bit addition

addw t4, t0,t1  #64-bit addition

addi a7, x0, 93

ecall
```

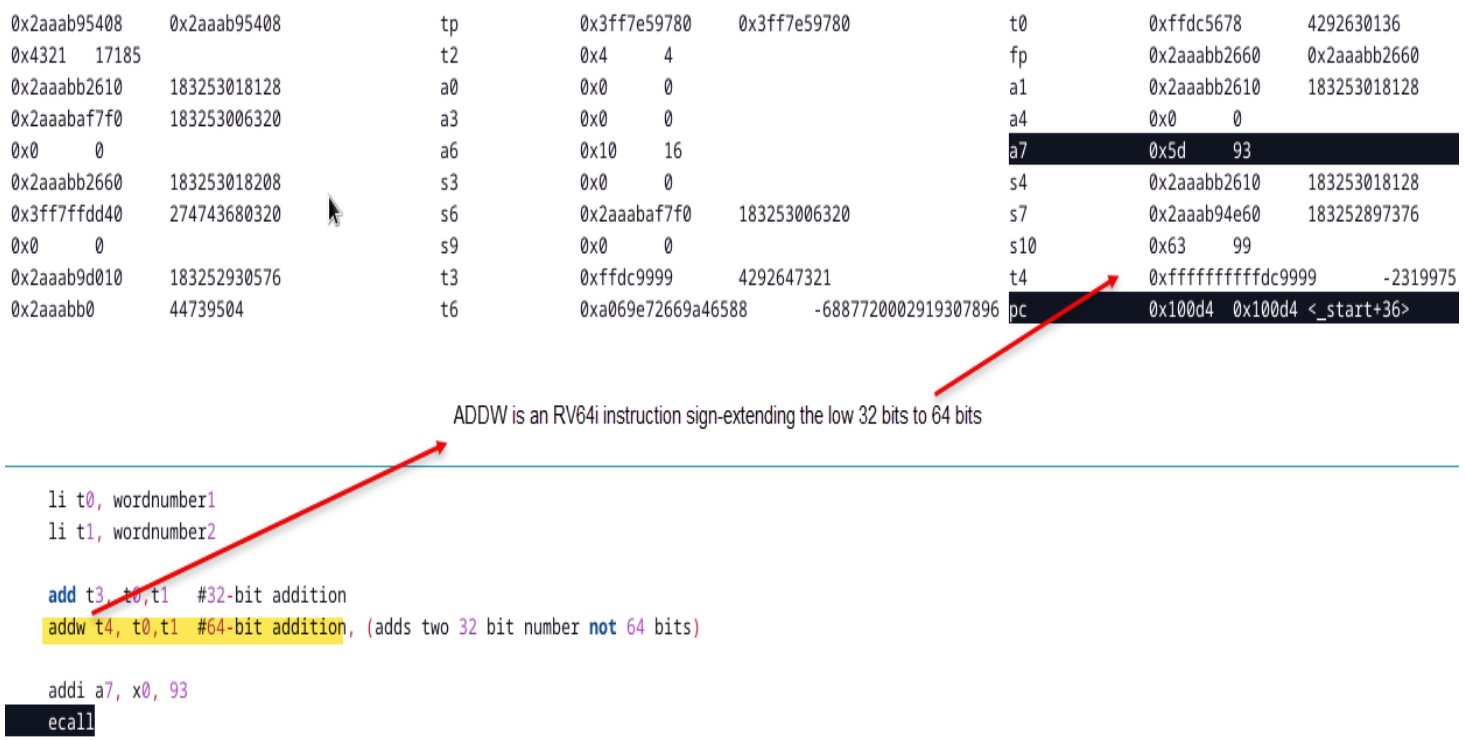


Figure 4-1 ADD and ADDW instructions

The instruction ADD gives a 32-bit result of 0xffdc9999

The instruction ADDW is an example of a 64-bit instruction which gives a 64-bit result.

Note the difference between ADD and ADDW.

- ADD takes bits 0-31 of each register and gives a 32-bit result
- ADDW takes bits 0-31 of each register and sign extends the addition into a 64-bit result.

The figure below shows that *sign extending*[36] 32-bit result ensures that the results are consistent.

[36] Sign extending is a technique of extending the most significant bit to preserve the sign and value of the number. Unsigned arithmetic will *zero extend* the high order bits so zero extending 16 bits to 32 bits gives 0X0045 → 0x00000045.

Table 4-2 Sign extension example

The disassembly for the .text section is as follows:

```
Disassembly of section .text:


00000000000100b0 <_start>:
100b0:      001002b7                lui   t0,0x100
100b4:      dc52829b                addiw t0,t0,-571        #        ffdc5
<__global_pointer$+0xee4ed>
100b8:      00c29293                slli  t0,t0,0xc
100bc:      67828293                addi  t0,t0,1656
100c0:      00004337                lui   t1,0x4
100c4:      3213031b                addiw t1,t1,801 # 4321 <wordnumber2>
100c8:      00628e33                add   t3,t0,t1
100cc:      00628ebb                addw  t4,t0,t1
100d0:      05d00893                addi  a7,zero,93
100d4:      00000073                ecall
```

The `li t0, 0xffdc5678` instruction breaks down into:

```
lui t0, 0x100
lddiw t0, t0, -571
slli t0, t0, 0xc
addi t0, t0, 1656
```

This results in t0 being equal to 0XFFDC5678 as shown in Figure 4-2.

Figure 4-2 Calculating LI to, 0xffdc5678 non-aliased steps

T0	31												20								11											0
LUI t0, 0x1	0x100000	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ADDIW t0,	0xffdc5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	0	0	1	0	1
SLLI t0, t0,	0xffdc5000	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
ADDI 1656	0xffdc5678	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	0

The instruction `SLLi` has not been met before. This instruction performs a left shift by the number of places in the immediate operand which means shift the value currently in t0 12 places to the left.

4.2.2. ADD Immediate

The ADDI (add immediate instruction) has the form `addi rd, rs, imm12`. The source register is added to a 12-bit immediate value and the result is placed in the destination register. The format of this instruction has already been described on page 2-8.

The ADDI instruction is straightforward –

Listing 4-2 ADDi example

```
* Listing 4-2 Simple addition (immediate) instruction */


.section .data

.equ wordnumber1, 0xffdc5678

.equ wordnumber2, 0x4321

.equ myfirstconstant, 0x1ff

.equ mysecondconstant, 0

.section .text

.global _start

_start:

li t0, wordnumber1

li t1, wordnumber2

addi t3, t0,myfirstconstant # t3 = 0xffdc5877

addiw t4, t1,mysecondconstant  # t4 = 0x4321

addi a7, x0, 93

ecall
```

4.2.3. MV instruction

The MV instruction is aliased to ADDI. The format is `mv rd, rs` as shown in Listing 4-3. After execution the contents of t0 will have been copied[37] to t1.

Listing 4-3 MV instruction

```
/* Listing 4-3

Move instruction. actually a pseudo instruction

mv rd, rs --> addi rd, rs, 0*/

.section .data

.equ number1, 0x12345678

.section .text

.global _start

_start:

li t0, number1

mv t1, t0

addi a7, x0, 93

ecall
```

The unaliased listing is

```
-<_start>:

100b0:      123452b7                lui   t0,0x12345

100b4:      6782829b                addiw t0,t0,1656 # 12345678 <number1>

100b8:      00028313                addi  t1,t0,0

100bc:      05d00893                addi  a7,zero,93

100c0:      00000073                ecall
```

SUB instruction

[37] The Move instruction is really a copy function, in that the source register's contents are preserved.

The available subtraction instructions are SUB and SUBW, there is no subtract immediate variant since this can be achieved through addition, by adding a negative number.

Listing 4-4 Use of SUB and SUBW instructions

```
/* Listing 4-4 Subtraction operations

32-bit (sub) and 64-bit (subw) are shown */

.section .data

.equ wordnumber1, 0xffdc5678

.equ wordnumber2, 0x4321

.equ negativenumber, -4

.section .text

.global _start

_start:

li t0, wordnumber1

li t, wordnumber2

sub t3, t0, t1  # 0xffdc1357; 32-bit positive result

sub t4, t1, t0  # 0xffffffff0023eca9; 32-bit negative result

subw t5, t0, t1 # 0xffffffffffdc357; 64-bit positive result

addi t2, t1, negativenumber #0x431d; subtracts 4 from t1 result --> t2

addi a7, x0, 93

ecall
```

4.3. Condition Codes

Many processors incorporate a condition code register (CCR) or status register to detect conditions such as

- Negative (N) True when signed number is negative, false if positive.
- Zero (Z) True if result such as comparison of values are equal, false if not equal.
- Carry (C) True If carry or no borrow condition occurs, shifted out bit
- Overflow (V) True if and overflow condition occurs.

This is important for processors that have limited register sizes. Checking for these conditions takes time and for many programs conditions such as overflow and carry will never occur. This is the case where there is a finite number of elements well below the maximum register data width size. A 32-bit register can hold over 4 billion positive integers which will not be exceeded when dealing with real-world objects such as inventory, staff, weather temperatures etc. Clearly it would be wasteful to check for additive carry conditions when new personnel are hired. There are, however, situations where these situations can occur and in the case of RISC-V this can be checked.

4.3.1. Detecting an oVerflow condition

An example of an *overflow* condition occurs when the data is too large to fit into a register. Consider a small eight-bit register which can hold *signed* values ranging from -128 to +127. Adding two positive numbers such as 0x50 and 0x40 results in 90 which is a negative number in signed eight-bit arithmetic.

Table 4-3 Detecting an overflow condition (signed)

50_{16} =	0 (Sign bit+)	1	0	1	0	0	0	0
40_{16} =	0 (Sign bit+)	1	0	0	0	0	0	0
+ = 90	1 (Sign bit-)	0	0	1	0	0	0	0

For unsigned the result of an addition should not be a number smaller than either of the operands.

Table 4-4 Detecting an overflow condition (unsigned)

73_{16} =	0	1	1	1	0	0	1	1
95_{16} =	1	0	0	1	0	1	0	1
+ = 90	0	0	0	0	1	0	0	0

Note the 9th bit has been discarded (fallen into the *bit bucket*)!

In general -

- For signed arithmetic - If the operands have the same sign but the result is a different sign, then overflow has occurred.
- For unsigned the addition should not be smaller than either of the operands

Other conditions such as Negative, Carry and Borrows can also be checked for by software rather than implementing dedicated registers.

4.3.2. RVM Instructions

The ADD and SUB instructions are part of the *Base Integer Set* (RVI). Multiply and Divide instructions belong to the optional *Multiply/Divide instruction set (RVM).*

4.3.3. Multiply Instructions

The MUL instruction has the format `mul rd, rs1, rs2.`

First of all, run the multiply instruction and variants on an RV32 machine.

Listing 4-5 Multiply instructions on RV32

```
/*Listing 4-5 32-bit Multiplication operations

This system ran on RV32 Simulation (CPUlator)*/

.section .data

# Define four 32-bit words

word1: .word 0xffffffff

word2: .word 0x0fffffff

word3: .word 0xffffffff

word4: .word 0x8

.section .text

.global _start

_start:

lw t0, word1

lw t1, word2

lw a0, word3

lw a1, word4

#RISC-V documents state to execute in the order of MULH, MULHU, MULHSU

first then MUL

mulh t3, t0, t1 # t3 = 0xffffffff Upper 32 bits (63:32)

mul t2, t0,t1   # t2 = 0xf0000001 lower 32 bits (31:0)

# Overall 64 bit result is 0xfffffffff0000001

mulh a2, a0, a1 # a2 = 0xffffffff

mul a3, a0, a1 # a3 =  0xfffffff8

# Overall 64 bit result is 0xfffffffffffffff8
```

```
# Unsigned multiply

mulhu a2, a0, a1 # a2 = 0x00000007

mul a3, a0, a1 # a3 =   0xfffffff8

# Overall 64 bit result is 0x7fffffff8


# First operand is signed, second operand is unsigned

mulhsu a2, a0, a1 # a2 = 0xffffffff

mul a3, a0, a1 # a3 =    0xfffffff8

# Overall result is 0xfffffffffffffff8

addi a7, x0, 93

ecall
```

Next run on a 64-bit system

Listing 4-6 Multiply instructions on RV64

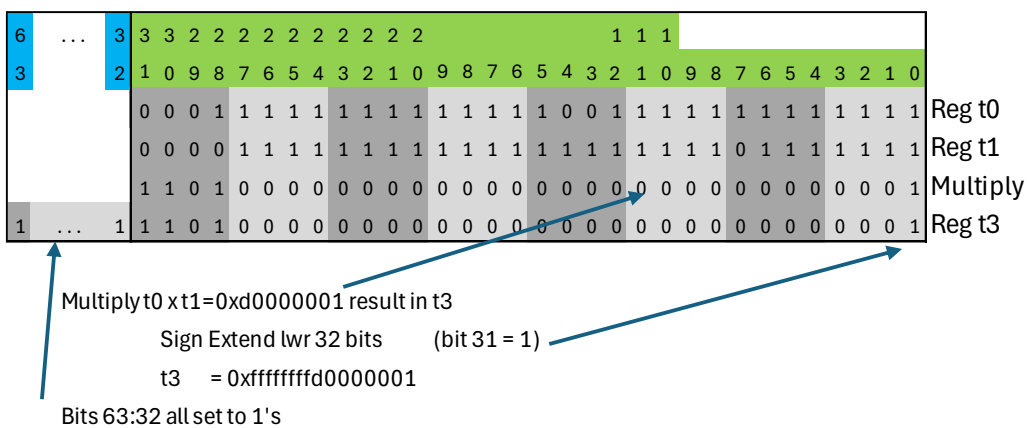
```
 /* Listing 4-6 Multiplication operations */

 .section .data

 # Define two 32-bit words

     word1: .word 0x1fffffff

     word2: .word 0x0fffffff

 # Define two 64 bit double words    doubleword1: .dword 0x0fffffffffffffff

#Leading zeros put in for clarity

     doubleword2: .dword 0x00000000f0000000

     doubleword3: .dword 0x8000000000000000

 .section .text

 .global _start

 _start:
```

```
  lw t0, word1

  lw t1, word29   ld a0, doubleword1

  ld a1, doubleword2

    ld a4, doubleword3

    # 32-bit product goes to t2

    mul t2, t0,t1 # t2 = 0x01ffffffd0000001

    mulw t3, t0, t1 # t3 =0xffffffffd0000001

    # 64-bit product

    mulh a2, a0, a1 # a2 = 0xefffffff

    mul a3, a0, a1  # a3 = 0xffffffff10000000

    mulhu a5, a4, a2 # a5 = 0x77fffff

    mul a6, a4, a2 # a6 = 0x8000000000000000

    addi a7, x0, 93

    ecall
```

MULH is used to get the upper 32 bits of the product.

The instruction MULW is a 64-bit instruction, multiplying the lower 32 bits of the source registers and sign extending bit 31 as shown in Figure 4-3. Combining the upper 32 bits from MUL and the lower from MULW gives the result 0x01ffffffd0000001

Figure 4-3 MULW instruction



In this case the product was sign extended. The next instruction where t0 has the value 0xfff and t1 has the value 0x8 gives a result of 0x7fff8 since the upper 32 bits from the MUL instruction equaled 0x0 and the lower 32 bits from the MULW instruction equaled 0x0007fff8.

To summarize –

Table 4-5 RVM Multiply Instructions

Instruction	Description	Data Polarity
MUL	Lower 32 bits of the product	
MULH	Upper 32 bits of the 64-bit product	Both operands are Signed
MULHU	Upper 32-bits of the 64-bit product	Both operands are Unsigned
MULHSU	Upper 32-bits of the 64-bit product	First operand is Signed, the second is Unsigned
MULW	Lower 32-bits of the product and sign- extend	Sign-extends to 64-bits

4.3.4. Divide Instructions

Division is simpler than multiplication, the instructions are DIV(W) (Divide Signed) , UDIV(W) (Divide Unsigned), REM (Remainder Signed) and REMU (Remainder unsigned). A basic example follows –

Listing 4-7 Division example

```
/*Listing 4-7 Division operations*/

.section .data

# Define two 32-bit words

word1: .word 1025

word2: .word 4

.section .text

.global _start

_start:

lw t0, word1

lw t1, word2

# Operand1 is the numerator

# Operand2 is the denominator

div t2, t0,t1   # t2 = 0x100
```

```
divu t3, t0,t1  # t3 = 0x100

rem t4, t0,t1   # t4 = 0x1

remu t5, t0, t1 # t5 = 0x1

addi a7, x0, 93

ecall
```

There are wide variants - DIVW and DIVUW are 64-bit instructions. These instructions divide the lower 32 bits of operand1 by the lower 32 bits of operand2. DIVW is for signed numbers and DIVUW are for unsigned. The result is sign-extended. The remainder instruction counterparts are REMW and REMUW also sign-extending to 64 bits.

4.3.4.1. Division by zero

Division by zero will generate a -1 result (all bits are set) and is not trapped. The remainder is equal to the dividend.. A further example is shown below:

Listing 4-8 Further Division examples

```
/*Listing 4-8 Further division operations*/

.section .data

# Define two bytes and a 32-bit word

word1: .word 0xffffffff1

byte1: .byte 1

byte2: .byte 4

.section .text

.global _start

_start:

lw t0, word1

lb t1, byte1

lb t2, byte2

mv t6, zero  # Use for division by zero

# Operand1 is the numerator

# Operand2 is the denominator
```

```
divw t3, t0, t1    # t3 = 0xfffffffffffffff1

remw t5, t0, t1    # t5 = 0x0

divuw t4, t0, t2   # t4 = 0x3ffffffc

remuw a0, t0, t2   # a0 = 0x1

# Divide by zero

div a1, t0, t6 # a1 = 0xffffffffffffffff

rem a2, t0, t6 # a2 = 0xfffffffffffffff1

divw a3, t0, t6 # a3 = 0xffffffffffffffff

remw a4, t0, t6 # a4 = 0xfffffffffffffff1

addi a7, x0, 93

ecall
```

Since division by zero gives the combination of all ones and the original dividend it can be checked after the division has taken place when necessary. The order given of DIV followed by REM in the listing is recommended for microarchitecture efficiency.

4.4. Shift Operations

RISC-V offers a number of shift instructions. Left shifts can be register or immediate. Shift Right instructions are similar except that they also offer a shift right arithmetic variant. This is summarized in Table 4-6.

Table 4-6 RV32 Shift Instructions

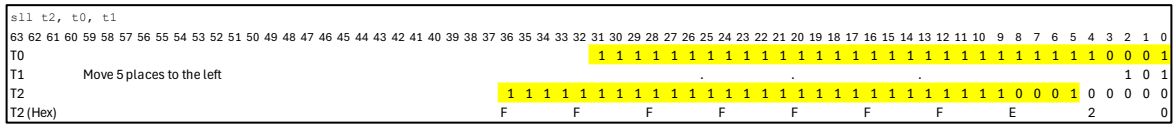
Instruction	Description	Syntax	Example
sll	Shift Left Logical, shifts rs1 bits leftwards by count in rs2 fills moved empty bit positions with zeros, result in rd.	`sll rd, rs1, rs2`	`sll t2, t0, t1`
slli	Shift Left Logical Immediate, shifts rs1 bits leftwards by count in immediate field, fills moved empty bit positions with zeros	`slli rd, rs1, imm`	`slli t3, t0, 18`
srl	Shift Right Logical, shifts rs1 bits rightwards by count in rs2 fills moved empty bit positions with zeros, result in rd	`srl rd, rs1, rs2`	`srl t4, t0, t1`

srli	Shift Right Logical Immediate, shifts rs1 bits rightwards by count in immediate field fills moved empty bit positions with zeros, result in rd	`srli rd, rs1, imm`	`srli t5, t0, 10`
sra	Shift Right Arithmetic, shifts rs1 bits rightwards by count in rs2 fills moved empty bit positions with the value of rs1's most significant bit, result in rd	`sra rd, rs1, rs2`	`sra t6, t2, t1`
srai	Shift Right Arithmetic Immediate, shifts rs1 bits rightwards by count in immediate field fills moved empty bit positions with the value of rs1's most significant bit, , result in rd	`srai rd, rs1, imm`	`srai t6,t2, 18`

RV64 features wide variants of the shift instruction which sign-extends to 64-bits – `slliw, srliw` and `sraiw.`

RV32 uses the five least significant bits (4:0) for the shift amount and RV64 will use the six least significant bits (5:0)[38]. Figure 4-4 shows a five-bit shift to the left, note that bits 4:0 are replaced by incoming zeros. As discussed earlier each move to the left is equivalent to multiplying by two. In this example the value 0xfffffff1 has been multiplied by 32 (2^5).

Figure 4-4 SLL instruction



Listing 4-9 Shift instructions example

```
/*Listing 4-9 Shift operations*/

.section .data

# Define data

word1: .word 0xfffffff1

byte1: .byte 0x5
```

[38] Using a larger value such as `srli t5, t0, 64` gives an error message such as "Error: improper shift amount (64)" by the GNU assembler.

```
.section .text

.global _start

_start:

lw t0, word1

lb t1, byte1

# Shift Left

sll t2, t0, t1      # t2 = 0xfffffffffe200000

slli t3, t0, 18    # t3 = 0xfffffffffc40000

# Shift Right

srl t4, t0, t1     # t4 = 0x7fffffffff

srli t5, t0, 10    # t5 = 0x3fffffffffffff

# Arithmetic shift right

sra t6, t2, t1     # t6 = 0xfffffffffffffff1

srai t6,t2, 18     # t6 = 0xffffffffffffff88

# Sign-extends to 64 bits

srliw t5, t0, 10   # t5 = 0x3fffff

addi a7, x0, 93

ecall
```

Currently there is no rotate instruction.

4.5. Logical Instructions

RISC-V includes the following family of logical instructions:

- AND
- OR
- XOR
- NOT

These instructions are summarized in Table 4-7.

Table 4-7 RISC-V Logical Instructions

Instruction	Description	Syntax	Example
AND	Performs rs1 and rs2 bitwise AND operation, placing result in rd	and rd, rs1, rs2	and a0, t0, t1
ANDI	Performs rs1 and sign-extended immediate field bitwise AND operation, placing result in rd	andi rd, rs1, imm	andi a1, a0, 0xf
OR	Performs rs1 and rs2 bitwise OR operation, placing result in rd	or rd, rs1, rs2	or a2, t0, t1
ORI	Performs rs1 and sign-extended immediate filed bitwise OR operation, placing result in rd	ori rd, rs1, imm	ori a3, a2, 0x000
XOR	Performs rs1 and rs2 bitwise XOR operation, placing result in rd	xor rd, rs1, rs2	xor a4, t0, t3
XORI	Performs rs1 and sign-extended immediate field bitwise XOR operation, placing result in rd	xor rd, rs1, imm	xori a5, a2, 0xa
NOT	Performs bitwise inversion of bits in rs1 placing result in rd	not rd, rs1	not a5, a5

Listing 4-10 and Listing 4-11 show the result of the various logical instructions on RV32 and RV64. The code is the same except for the comments which show how RV64 sign-extends the results.

Listing 4-10 Logical Instructions (RV32)

```
/*Listing 4-10 Logical operations RV32 system*/

.section .data

# Define data

word1:      .word 0xaa55aa55

maskupper: .word 0xfff

masklower: .word 0x0

xormask1:   .word   0xaaaaaaaa

xormask2:   .word   0x55555555

.section .text

.global _start
```

```
_start:
lw t0, word1 # t0 sign-extended = 0xaa55aa55
lw t1, maskupper
lw t2, masklower
lw t3, xormask1 # t3 sign-extended = 0xaaaaaaaa
lw t4, xormask2
# AND
and a0, t0, t1     # a0 =  0xa55
and a0, t0, t2     # a0 = 0
andi a1, a0, 0xf   #a1 = 0
# OR
or a2, t0, t1     # a2 = 0xaa55afff
or a2, t0, t2     # a2 = 0xaa55aa55
ori a3, a2, 0x000 #a3 =  0xaa55aa55
# XOR
xor a4, t0, t3     # a4 = 0x00ff00ff
xor a4, t0, t4     # a4 = 0xff00ff00
xori a5, a2, 0xa   # a5 = 0xaa55aa5f
# NOT
not a5, a5 # a5 = 0x55aa55a0
addi a7, x0, 93
ecall
```

Listing 4-11 Logical Instruction (RV64)

```
/*Listing 4-11 Logical operations (RV64 system)*/
```

```
.section .data

# Define data

word1:     .word 0xaa55aa55

maskupper: .word 0xfff

masklower: .word 0x0

xormask1:  .word  0xaaaaaaaa

xormask2:  .word  0x55555555

.section .text

.global _start

_start:

lw t0, word1 # t0 sign-extended = 0xffffffffaa55aa55

lw t1, maskupper

lw t2, masklower

lw t3, xormask1 # t3 sign-extended = 0xffffffffaaaaaaaa

lw t4, xormask2

# AND

and a0, t0, t1     # a0 =  0xa55

and a0, t0, t2     # a0 = 0

andi a1, a0, 0xf   #a1 = 0

# OR

or a2, t0, t1     # a2 = 0xffffffffaa55aa55

or a2, t0, t2     # a2 = 0xffffffffaa55aa55

ori a3, a2, 0x000 #a3 =  0xffffffffaa55aa55

# XOR

xor a4, t0, t3     # a4 = 0xff00ff
```

```
xor a4, t0, t4      # a4 = 0xfffffffff00ff00

xori a5, a2, 0xa    # a5 = 0xffffffffaa55aa5f

# NOT

not a5, a5 # a5 = 0x55aa55a0

addi a7, x0, 93

ecall
```

4.5.1. Logical function observations

- The AND function can be used to clear bits by *anding* the corresponding bit position with a binary zero.
    - Bits can be tested to see if they are high or low by anding with a binary one.
        - A non-zero value denotes that the corresponding bit tested was a binary one
        - A zero value denotes that the corresponding bit tested was a binary zero
- Bits can be set by *oring* the corresponding bit position with a binary one
    - Bits can be tested to see if they are high or low by oring with a binary zero
        - A non-zero value denotes that the corresponding bit tested was a binary one
        - A zero value denotes that the corresponding bit tested was a binary zero
- Exclusive or can check to see if the corresponding bit has equal polarity
    - A non-zero value indicates that the bit was of the opposite polarity
    - A zero value indicates that the bit was if the same polarity
    - Applying the exclusive or function using the same bit pattern as the number itself will clear the bits

Exercises for chapter 4

1. Write code to perform multiplication by 24 using shift instructions, do not use RISC-V multiply instruction variants

RISC-V instructions covered in chapter 4

Arithmetic Instructions (Base ISA)

- **add** – Add (32-bit)
- **addw** – Add word (sign-extended to 64-bit)
- **addi** – Add immediate
- **sub** – Subtract (32-bit)
- **subw** – Subtract word

Multiply and Divide Instructions (RVM Extension)

- **mul** – Multiply
- **mulh** – Multiply high (signed × signed)
- **mulhsu** – Multiply high (signed × unsigned)
- **mulhu** – Multiply high (unsigned × unsigned)
- **mulw** – Multiply word (32-bit)
- **div** – Divide (signed)
- **divu** – Divide unsigned
- **rem** – Remainder (signed)
- **remu** – Remainder unsigned
- **divw** – Divide word
- **remw** – Remainder word

Shift Instructions

- **sll** – Shift left logical
- **srl** – Shift right logical
- **sra** – Shift right arithmetic
- **sllw** – Shift left logical word
- **srlw** – Shift right logical word
- **sraw** – Shift right arithmetic word

Logical Instructions

- **and** – Bitwise AND

- **or** – Bitwise OR
- **xor** – Bitwise XOR
- **andi** – AND immediate
- **ori** – OR immediate
- **xori** – XOR immediate
- **not** – Bitwise NOT (pseudo-instruction)

Chapter 5. Loops, Branches and Conditions

Overview of the chapter

Chapter 5 introduces control flow mechanisms in RISC-V assembly. It explains how to make decisions, repeat code (loops), and redirects program execution using conditional and unconditional branches mechanisms.

5.1. J-Type and B-Type instructions

Paragraphs 2.2.1.3.4 and 2.2.1.3.5 discussed the control transfer J and B type instructions. Recall that unconditional jumps are J-type and conditional branches are B-type. The ability to vary the program flow, based on conditions such as greater than (>), less than (<) or equality greatly enhances the power of computing devices. RISC-V can perform conditional branches with a single instruction. Other instruction sets may use two instructions, by first performing a comparison and then deciding whether to branch by the status of a condition code register flag.

Comparison using two instructions -

```
cmp r1, r2 # Compare two registers

bgt <label> # Branch if the value of register1 is greater than the value

of register2
```

RISC-V only uses a single instruction -

```
bgt t0, t1, <label> # Branch if t0 is less than t1
```

This can result in more economic code.

5.1.1. B-Type instruction details

Consider the instruction `blt t0, t1, exit` where the branch instruction is located at 0x100c0 and the label <exit> is located at location 0x100c8. When t0 is less than t1 then the flow will branch to the address at <exit>. The opcode in this case is 0x0062c463.

Referring to Figure 5-1 the diagram shows that the offset has a value of 8 which is the number of places that the program will branch to (0x100c8 minus 0x100c0). Recall that bit zero need not be encoded in the immediate value which specifies the offset and is always implicitly set to zero. This means that the offset is always even. The reason that the offset is a multiple of two rather than four is to accommodate RISC-V 16-bit implementations. The register operands use the X register number showing the values 5 and 6 which correspond to registers t1 and t0.

Figure 5-1 Breakdown of blt instruction

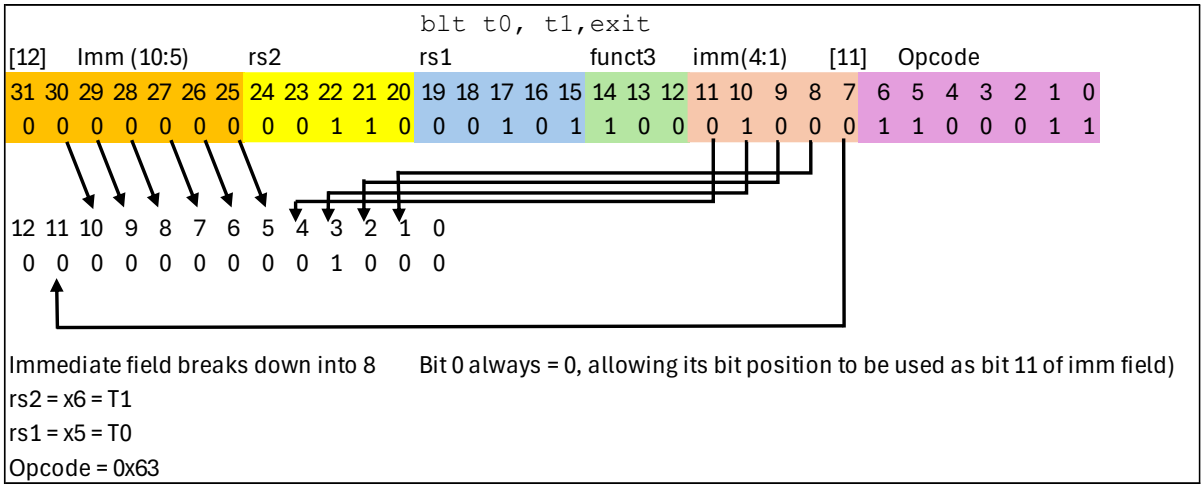


Table 5-1 shows the available branch instructions which includes the additional pseudo instructions.

Note that comparisons can be made with signed and unsigned values.

Table 5-1 Conditional branch instructions

Instruction	Description	Syntax	Example
blt	Branch if less than	blt rs1, rs2, imm	blt t0, t1, exit[39]
bltu	Branch if less than unsigned	bltu rs1, rs2, imm	bltu t0, t1, exit
bltz	Branch if less than zero*	bltz rs1, imm	bltz t0, exit
ble	Branch if less than or equal to zero*	ble rs1, rs2, imm	ble t0, t1, exit
bleu	Branch if less than or equal unsigned*	bleu rs1, rs2, imm	bleu t0, t1, exit
blez	Branch if less than or equal to zero*	blez rs1, imm	blez t0, exit
bge	Branch if greater than or equal	bge rs1, rs2, imm	bge t0, t1, exit
bgt	Branch if greater than*	bgt rs1, rs2, imm	bgt t0, t1, exit
bgtu	Branch if greater than unsigned*	bgtu rs1, rs2	bgt t0, t1, exit
bgtz	Branch if greater than zero*	bgtz, rs1, imm	bgtz t0, exit
bgeu	Branch if greater than or equal to zero unsigned	bgeu rs1, rs2, imm	bgeu t0, t1, exit
bgez	Branch if greater than or equal to zero*	bgez rs1, imm	bgez t0, exit
beq	Branch if equal	beq rs1, rs2, imm	beq t0, t1, exit

[39] This is a memory location pointed to by the label “exit”

Instruction	Description	Syntax	Example
beqz	Branch if equal to zero*	beqz rs1, imm	beqz t0, exit
bne	Branch if not equal	bne rs1, rs2, imm	bne t0, t1, exit
bnez	Branch if not equal to zero*	bnez rs1, imm	bnez t0, exit

*=Pseudo instruction

5.1.2. J-Type instruction details

5.1.2.1. JAL

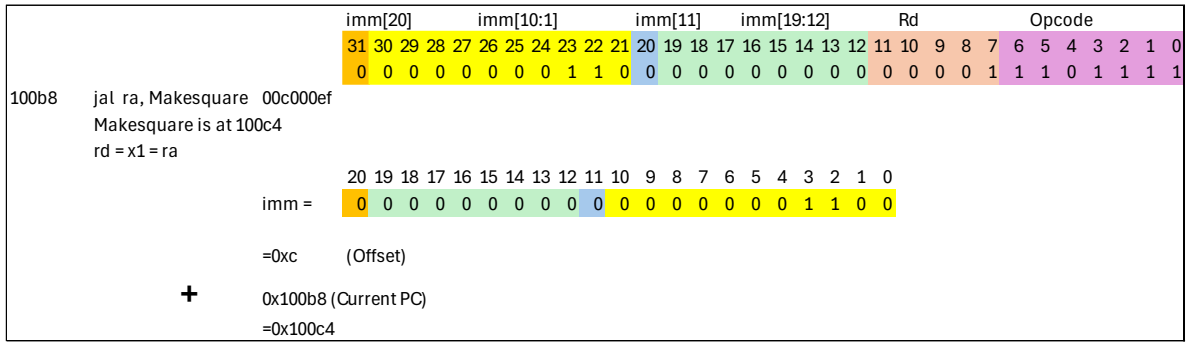
The format of the Jump and link instruction (`JAL`) is `JAL rd, <label>`.

The instruction `jal Makesquare` is equivalent to the pseudo instruction `j Makesquare` which disassembles to the non-aliased instruction `jal ra,<Makesquare>.` It has a 20-bit immediate value specifying bits 20:1. Bit 0 of the immediate value is not coded and always set to zero to give even values. This gives a total of 21 signed bits which is equivalent to a range of minus one MB through to plus one MB.

By convention rd is register X1 (ra), if no destination register is specified then ra is automatically used. An instruction such as `jal ra, Makesquare` will use an offset to the address located at the label <Makesquare>. The return address register (ra) will hold the address of the next instruction following the current `jal ra, Makesquare` instruction (current PC+4 = 0x100bc).

Referring to Figure 5-2 the immediate value is 0xc so adding this value to the address of the JAL instruction (here 0x100b8) gives a jump address of 0x100c4 which is where the Makesquare routine is located. When the routine has finished the program flow returns to the address stored in the ra register (0x100bc). This is achieved by the pseudo instruction `ret` which is an alias for `jalr, zero,0(ra).`

Figure 5-2 Bit breakdown of JAL instruction



5.1.2.2. JALR

The jump and link register instruction (JALR) gets its target address by adding a sign-extended 12 -bit value to the source register rs1, setting the least significant bit to zero. The destination register will be loaded with the address of the instruction following the JALR instruction address.

5.2. Implementing a loop counter to square numbers

The first example is that of a simple loop counter. The program uses a sub-routine to compute squares of numbers from 1 to 20. The results are stored in consecutive halfword locations. The listing features one unconditional branch (`blt`) and one unconditional jump (`jal`). After the sub-routine has completed the `jalr` instruction will jump to the instruction (`addi a7,zero,93`) immediately following the instruction (`jal squareit`) that called the sub-routine.

Listing 5-1 Squaring numbers from 1 to 20

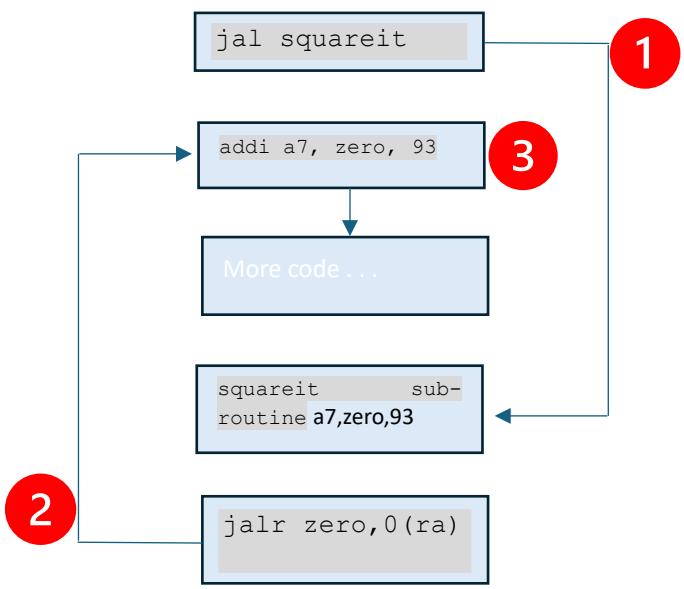
```
# listing5-1.s

section .text

.global _start

_start:

addi  t0,zero,21  # Set up counter

addi  t1,zero,1 # Start at 1

la a0, storesquares

jal squareit # Jump to routine at <squareit>, saving return address in ra

addi  a7,zero,93 # Routine finished, time to leave

ecall

squareit:

mul   t2,t1,t1 # Square the contents of t1 and put the result in t2

sh t2,0(a0)

addi a0, a0,2 # Point to the next halfword location (2 bytes on)

addi  t1,t1,1 # increment the number to be squared

blt   t1,t0,squareit # If 20 numbers have been squared then return from

routine

jalr  zero,0(ra)

.section .data

storesquares:
```

```
.space 64
```

Examining memory after the sub-routine has finished shows -

```
0x1111c:    0x0001      0x0004      0x0009      0x0010      0x0019  0x0024
      0x0031      0x0040
0x1112c:    0x0051      0x0064      0x0079      0x0090      0x00a9
      0x00c4      0x00e1      0x0100
0x1113c:    0x0121      0x0144      0x0169      0x0190
0x1111c:    1     4     9     16    25    36    49    64
0x1112c:    81    100   121   144   169   196   225   256
0x1113c:    289   324   361   400
```

Figure 5-3 Program flow of Makesquares listing



1. Call sub-routine <squareit>
2. Return from sub-routine
3. Resume code execution

Exercises for chapter 5

1. Write a program that takes as its input a number less than 1000 and calculate the number of primes below that number.

RISC-V instruction covered in chapter 5

B-Type (Conditional Branches)

blt – Branch if less than

bltu – Branch if less than unsigned

bltz – Branch if less than zero *(pseudo-instruction)*

ble – Branch if less than or equal *(pseudo-instruction)*

bleu – Branch if less than or equal unsigned *(pseudo-instruction)*

blez – Branch if less than or equal to zero *(pseudo-instruction)*

bge – Branch if greater than or equal

bgt – Branch if greater than *(pseudo-instruction)*

J-Type (Unconditional Jumps)

jal – Jump and link

jalr – Jump and link register

Chapter 6. The Stack, Macros and Functions

Overview of the chapter

*Chapter 6 focuses on **modularizing code** in RISC-V assembly using **functions, macros, and the stack**. It introduces structured programming principles in low-level development and shows how to organize code effectively.*

6.1. Overview

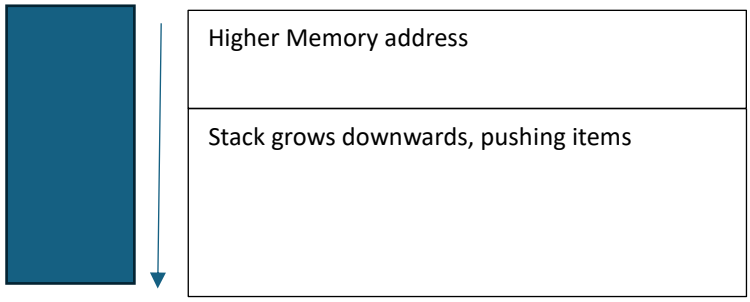
The concepts between macros and functions are similar but the way that the programs are assembled leads to tradeoffs behind performance and code size. The previous chapter used a sub-routine called `<squareit>`. The routine can be a separate piece of code outside of the main listing which means that routines can be used as functions that other programs can call on rather than having to keep writing the additional code enhancing clarity and manageability.

6.1.1. The Stack

Functions will make use of the stack. The stack is a data structure which stores data in a structured manner. As an example, a register's contents can be *Pushed* on to the stack and can be restored by *Popping* the data from the stack back to the register again. Push and Pop operations are performed in a *Last in First out (LIFO)* manner, in that if multiple items were pushed on to the stack the last item pushed would be the first one restored. The stack is a location in memory. The *stack pointer* will show where in memory the top of the stack is situated. When data is pushed the stack pointer will be decremented to a lower memory location and when data is popped, the stack pointer will be incremented. The stack by default with RISC_V grows downwards.

A push to the stack is accomplished using the store instruction and a pop is accomplished using the load instruction. Both these instructions are familiar, the only difference being that the stack pointer is used as the operand rather than a normal register. With RISC-V the convention is to use register X2 as the stack pointer, its ABI name is sp. RV64 architectures require that the stack must be 16-byte (128-bits) *aligned*.

Figure 6-1 Stack contents operations



Functions are used to promote coding efficiency and clarity. They are sections of code that can be included in a program and shared with others as *libraries*. Over time a coder will usually generate their own functions for use in their code. When using external functions, registers can be saved on the stack prior to calling the function, thus ensuring that on return from the function code everything has been restored, and coding will continue from where it left off. The Program Counter (PC) keeps track of the location in memory where the

code is next to be executed. When a portion of code calls a function, it is termed the *caller*. The code that was called (the function itself) is termed the *callee*. When calling a function there are several tasks that the caller must perform and similarly the callee has its own responsibilities. When a function calls another function then the ra register must be preserved otherwise the original return address used by the first calling routine will be lost.

The registers follow certain conventions which are described below:

1. There are eight argument registers a0-a7
2. Additional arguments are pushed onto the stack and popped by the called routine.
3. Two registers are used for return values – A0 and A1
4. More values will use a reference to an address (call by reference) where the additional data is stored.
5. Values equal to double the XLEN bits can be passed using two registers. The low order XLEN bits are passed the lowest number register such as a0 and the high order XLEN bits passed in the higher register such as a1
6. The value can be passed on the stack.
7. If there is only one register available then it can be used, in conjunction with the stack.
8. Nested functions *must* preserve the ra register,
9. *Leaf* functions[40] do not need to save the return address to the stack,
10. When functions are called without knowing the register usage then the rules shown in Table 2-3 must be respected[41].

6.1.2. Combining separate programs

The first program (`maina.s`) calls an external program (`squareit.s)` to calculate the squares.

Listing 6-1 main.s

```
.section .data

message:          .ascii "\nPlease enter a sequence of digits (up to 4

characters)to be squared\n"
```

[40] A leaf function is a function that has been called but does not call any other functions.

[41] A summary of the rules –

Zero register (x0) is immutable,

ra must be preserved,

sp Caller must interact with via push and pop,

t0-t7 If needed should be saved by the calling routine,

s0-s11 Saved by the callee if used via the stack,

a0-a7 If needed should be saved by the calling routine,

```
.equ messagelength, 70
errormessage1:   .ascii "\nIllegal character(s) found, please enter only
base10 numbers\n"
.equ errormessage1length, 63
errormessage2:  .ascii "\nIncorrect number of digits, please enter 1,2,3
or 4 digits\n"
.equ errormessage2length, 60
inputbuffer:    .space 16 # Holds user input
numberbuffer:   .space 16 # Holds the converted ASCII to integer numbers
asciioutputbuffer: .space 16
successmessage: .ascii "\nThe result is "
.equ successmessagelength, 16
tidyupchars: .ascii "\n\n"
.equ tidyupcharslength, 2
.equ linefeed, 10
.section .text
.global _start
_start:
# Prompt for number
li a0, 1 #<stdout>
la a1, message
li a2, messagelength
li a7, 64
ecall
# Read in number from keyboard
li a0, 0 # file descriptor 0 (stdin)
```

```
la a1, inputbuffer # address of the buffer
li a2, 5 # Max number of bytes to read
li a7, 63 # Read syscall
ecall
# Convert inputted ASCII number to decimal
# Load the address of the ASCII number string
la a0, inputbuffer      # a0 = address of "string"
li a1, 0                # a1 = result (initialize it to 0)
li t0, linefeed # Linefeed character
li a3, 48        # Ascii number is actual number +48 so need to subtract
this value
li a4, 10         # used in convert_loop to multiply input character to
correct position
li a5, 57       # upper bound (entered digit can't be > 9)
li a6, 48       # lower bound (entered digit can't be < 0)
la t5, inputbuffer+5    # Check for too many digits entered
convert_loop:
# Load the next ASCII character
lb a2, 0(a0)            # a2 = *a0 (current ASCII character)
beq t0, a2,skip_valuechecks #(if <LF> then skip the checks for legal decimal
number)
bgt a2,a5,illegalcharacter  #(too high)
ble a2, a6,illegalcharacter #(too low)
skip_valuechecks:
# Check if we've reached the <LF> character (end of input string)
```

```
beq t0, a2, conversion_done # If character is <LF> then all numbers have
been processed
# Convert ASCII to integer:
li a3, 48         # Load '0' ASCII value
sub a2, a2, a3    # Convert ASCII character to integer
mul a1, a1, a4   # shift left by one decimal place
add a1, a1, a2  # Add the digit to result
# Move to the next character in the string
addi a0, a0, 1   # Increment the pointer
beq t5,a0, toomanydigits # More than 4 digits have been entered
j convert_loop # Next character
conversion_done:
mv a0,a1
jal squarenumber
# The number has been squared, time to convert back to ASCII format
la a1, asciioutputbuffer +11   # Point to the end of buffer
sb zero, 0(a1)        # Null-terminate the string
li t3, 0
convertbacktoascii:
addi t3, t3, 1
li t1, 10             # Load divisor (10)
rem t2, a0, t1        # Get last digit (a0 % 10)
div a0, a0, t1        # Remove last digit (a0 / 10)
addi t2, t2, 48       # Convert digit to ASCII (for printing)
addi a1, a1, -1       # Move buffer pointer back one place
sb t2, 0(a1)          # Store ASCII character in buffer
```

```
bnez a0, convertbacktoascii # Repeat if number is not zero


printsuccess:
li a0, 1              # syscall for print_string
la a1, successmessage         # Address of ASCII string
li a2, successmessagelength
li a7, 64              # Syscall number for printing string
ecall                 # Make syscall


print:
li a0, 1              # syscall for print_string
la a1, asciioutputbuffer             # Address of ASCII string
li a2,12
li a7, 64              # Syscall number for printing string
ecall                 # Make syscall
li a0, 1
la a1, tidyupchars
li a2, tidyupcharslength
li a7, 64
ecall
exit:
li a7, 93             # Syscall for exit
ecall
illegalcharacter:
li a0, 1
la a1, errormessage1
```

```
li a2, errormessage1length

li a7, 64

ecall

j exit

toomanydigits:

li a0, 1

la a1, errormessage2

li a2, errormessage2length

li a7, 64

ecall

j exit
```

Listing 6-1 squareit.s

```
. .global squarenumber

squarenumber:

# Note Register a0 contains the user input (the number to be squared)

# The same register will hold the return value

mul   a0,a0,a0 # Square the contents of a0 and put the result in a0

jalr  zero,0(ra)
```

The `makefile` is :

Listing 6-2 Makefile example using two programs

```
OBJECTS = maina.o squareit.o

all:maina

%.o:%.s

as -mno-relax $^ -g -o$@

maina:$(OBJECTS)

ld -o maina $(OBJECTS)
```

Validate the program –

```
./maina
Please enter a sequence of digits (up to 4 characters)to be squared
6
The result is
36
./maina
Please enter a sequence of digits (up to 4 characters)to be squared
12
The result is
144
./maina
Please enter a sequence of digits (up to 4 characters)to be squared
843
The result is
710649
./maina
Please enter a sequence of digits (up to 4 characters)to be squared
9999
The result is
99980001
$ ./maina
Please enter a sequence of digits (up to 4 characters)to be squared
3f
Illegal character(s) found, please enter only base10 numbers
./maina
```

```
Please enter a sequence of digits (up to 4 characters)to be squared

23146

Incorrect number of digits, please enter 1,2,3 or 4 digits
```

Note that the *called* program `<squareit.s>` has declared <squarenumber> as a *global*, this is to allow it to be shared and used by other files. Declarations without the .global directive are treated as *local* to the file that they were declared in and not accessible by other programs.

When tracing the program flow with GDB set the first breakpoint to `b _start` rather than `b 1`.

In the case of Listing 6-1 and Listing 6-1, the programs are named `maina.s` and `squareit.s.` The make file is:

```
OBJECTS = maina.o squareit.o

all:maina

%.o:%.s

as -mno-relax $^ -g -o$@

maina:$(OBJECTS)

ld -o maina $(OBJECTS)
```

The program flow is:

1. Prompt user to enter a number
2. Get the number from keyboard entry, storing it in *inputbuffer*
3. Convert the number from ASCII representation to integers, storing it in *numberbuffer*
4. Validate the number to be in the range <0-9> unless the ASCII character is Linefeed[42], If not valid then print out an error message (errormessage1) and exit.
5. Validate the quantity of digits entered, only 1,2,3 or allowed, if invalid then print out an error message (errormessage2) and exit.
6. The convert_loop routine will place the converted integers in the correct buffer location, the multiplication by ten shifts the digit to the correct magnitude value.[43]
7. Once the <linefeed> character has been encountered then all digits have been converted.

[42] Pressing enter on the keyboard will store the linefeed character (0xa) in the buffer

[43] For example the number 6543 will be processed in stages as 6, 60, 65, 650, 654, 6540, 6543 by the *multiply* and *add* instructions in convert_loop

8. The number is passed to the squarenumber routine in the program squareit. The squared number will be returned via register a0[44].

9. The next step is to display the result by converting the squared integer back to ASCII format which is performed by the routine convertbacktoascii.

10. Finally, the result is printed to the screen and the program exits.

The program should be stepped through with GDB; ensure that the routines - `convert_loop`: and `conversion_done`: are fully understood.

6.2. Macros

Macros, like functions can be used to promote coding efficiency and clarity. Macros can be included inline within a program or defined separately sing the `.include` directive. Macro code is encased between the directives .`macro` and .`endm`. They are used to repeat frequently used instructions using different parameter values. The format of a macro is macroname argument1, argument2, . . . Inside the macro code, these arguments have a backslash \ character in front of them/ Macro differ importantly from functions in that the actual macro code is substituted inline within the main code, this means that 100 calls to the same macro will generate 100 copies of the macro code. The use of Macro's can increase performance since there is no need to deal with return addresses as would be the case for functions.

Listing 6-3 Macro example (callmacro.s)

```
# This code calls a macro to print strings to stdout

# The input parameters are the string's location and its length

.section .data

string1:   .ascii "\nThis string was printed using a macro call"

string2:   .ascii "\nAnd so was this\n"

.equ stringlength1, 43

.equ stringlength2, 17

.section .text

.include "printmacro.s"

.global _start

_start:

li a0, 1 #stdout
```

[44] It has not been necessary to store values on the stack in this particular example. If the *called* routine were to overwrite any register that would need to be preserved then the caller/callee conventions would be respected.

```
# Save a0 on to the stack, would have been simpler to just load it again
after the macro call
# however this illustrates an example
# Allocate space on the stack
addi sp, sp, -16
sw a0, 12(sp)
print string1, stringlength1
lw a0, 12(sp)
# No need to preserve a0 this time since we no longer need to restore it
print string2, stringlength2
# Exit program
li a7, 93            # Syscall number for exit
ecall                # Make syscall
```

Listing 6-4 called macro program (printmacro.s)

```
.macro print location, length
la a1, \location
li a2, \length
li a7, 64
ecall
.endm
```

The disassembly (below) shows that the macro has been placed in line, GDB shows that the address of string1 is located at 0x11128 and that string2's address is at 0x11153.

```
(gdb) info variables
All defined variables:

Non-debugging symbols:
0x0000000000011128  __DATA_BEGIN__
0x0000000000011128  string1
0x0000000000011153  string2
0x0000000000011164  __SDATA_BEGIN__
0x0000000000011164  __bss_start
0x0000000000011164  _edata
0x0000000000011168  __BSS_END__
0x0000000000011168  _end
(gdb)
```

The next part of the macro is the `li` instruction which loads the string length into register a2, finally the syscall is invoked.

Note unlike functions there are no return calls since the macro code is inline.

```
00000000000100e8 <_start>:

100e8:       00100513                li      a0,1

100ec:       ff010113                addi    sp,sp,-16

100f0:       00a12623                sw      a0,12(sp)

100f4:       00001597                auipc   a1,0x1

100f8:      03458593              addi    a1,a1,52 # 11128 <__DATA_BEGIN__>

100fc:       02b00613                li      a2,43

10100:       04000893                li      a7,64

10104:       00000073                ecall

10108:       00c12503                lw      a0,12(sp)

1010c:       00001597                auipc   a1,0x1

10110:       04758593                addi    a1,a1,71 # 11153 <string2>

10114:       01100613                li      a2,17

10118:       04000893                li      a7,64

10120:       05d00893          li      a7,93

10124:      00000073          ecall
```

Exercises for chapter6

1. What is the purpose of the Link register?
2. Modify the program `maina.s` to keep running after an error message or successful result has been printed by asking the user if they would like to input another value (or not)
3. Delete the lines `sw a0, 12(sp)` and `lw a0, 12(sp)` from the `callmacro.s` program and comment on the program output.
4. Why is there an offset of 12 in the instruction `sw a0, 12(sp)?`
5. Explain the difference between a function and a macro
6. Which directives signify the start and end of a macro?
7. When is the `.include` directive used?

Summary of instructions used in chapter 6

Stack Management Instructions

addi – Used to adjust the stack pointer (sp)

Example: addi sp, sp, -32 (allocate stack space)

sd – Store doubleword (store register onto stack)

ld – Load doubleword (retrieve register from stack)

Control Transfer / Function Support

jal – Jump and Link (used to call functions)

ret – Return from function (pseudo-instruction for jalr x0, ra, 0)

jalr – Jump and Link Register (used indirectly via ret)

Macros and Utilities

.macro / .end_macro – Assembler directives for defining macros (not instructions but critical to macro usage)

Chapter 7. RISC_V assembly and C together

Overview of the chapter

Chapter 7 explores how assembly language and C code can work together, bridging low-level and high-level programming. It's highly practical for system developers who want to embed performance-critical routines in C-based applications.

7.1. Calling assembly functions from a high-level language

The first example creates two source programs, one written in C[45] code and the other in RISC_V assembly. The C program (`listing 7-1.c`) declares an external function (`getproduct`), located in the assembly program (`listing7-1.s`) and calls it passing the two arguments via a0 and a1. It then calls the `printf` function to output the result. The assembly program is executed in the normal fashion generating an object file. The `gcc` program[46] generates the C object file and links it with the previously generated object file.

Listing 7-1 RISC-V multiply function called from C

```
cat listing 7-1.c

/* This code shows how to call an assembly language program from C*/

#include <stdio.h>

// Declare the assembly function

extern int getproduct(int a, int b);

int main() {

int x = 100, y = 200;

// Call RiscV assembly function

int result = getproduct(x, y);

printf("The product of %d and %d is %d\n", x, y, result);

return 0;

}

cat listing7-1.s
```

[45] None of the c code presented here is overly complex, however if the reader is not familiar with C, there is a wealth of on-line tutorials to be consumed that will cover the basics for what is needed here.

[46] An alternative c compiler is `clang` which can be installed by `sudo apt install -y clang.`

```
.text

.global getproduct

getproduct:

mul a0, a0, a1  # Add the two input registers (a0 and a1) and store in a0

ret             # Return
```

The commands to generate the output file (listing 7-1) are:

```
$ as -g -o listing7-1.o listing7-1.s

$ gcc -o listing7-1 listing7-1.c listing7-1.o
```

Here `gcc (`Gnu complier collection) is used instead of the `ld` command that was previously used to perform the linkage.

The generated assembly files from the `.c` listing can be saved during compilation with the option `-save-temps`. Alternatively to just generate the RISC_V assembly code use the command `gcc -S <filename.c>` which generates `<filename.s>`

The command line is:

```
gcc -save-temps -o listing7-1 listing7-1.c listing7-1.o
```

This generates a file `listing7-1-listing7-1.s`. The bolded and italicized comments have been added to help with explanation and were not part of the `-save-temps` output.

```
$ cat listing7-1-listing7-1.s

.file "listing7-1.c"

.option pic

.attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"

.attribute unaligned_access, 0

.attribute stack_align, 16

.text

.section    .rodata

.align      3

.LC0:

.string     "The product of %d and %d is %d\n" # String as defined as part

of the C source
```

```
.text
.align      1
.globl      main
.type main, @function
main:
addi  sp,sp,-32   # Allocate space on the stack
sd    ra,24(sp)   # Store a double word (64 bits, our architecture is RV64)
from the ra register with an offset of 24 from the stack pointer
sd    s0,16(sp)   # Store a double word from the s0(fp) register with an
offset of 16 from the stack pointer
addi  s0,sp,32    # Store stack pointer with an offset of 32, from the
current stack pointer
li    a5,100            # First factor
sw    a5,-20(s0)  # Store first factor
li    a5,200            # Second factor
sw    a5,-24(s0)  # Both factors stored in addresses pointed to by s0 (s0
-20, s0 024)
lw    a4,-24(s0)  # Load a4 with second factor
lw    a5,-20(s0)  # Load a5 with first factor
mv    a1,a4       # Move second factor to a1
mv    a0,a5       # Move first factor to a0
call  getproduct@plt # Call function with parameters held in a0 and a1
mv    a5,a0       # Move first factor into a5
sw    a5,-28(s0)  # Store first factor into location pointed to by s0 with
an offset of -28
lw    a3,-28(s0)  # Load first factor into a3
```

```
lw    a4,-24(s0)  # Load second factor into a4
lw    a5,-20(s0)  # Load first factor into a5
mv    a2,a4       # Load second factor into a2
mv    a1,a5       # Load first factor into a1
lla   a0,.LC0
call  printf@plt  # call printf function using the procedure linkage table[47]
li    a5,0
mv    a0,a5
ld    ra,24(sp)   # Pop ra register back to original value
ld    s0,16(sp)   # Pop frame pointer back to original value
addi  sp,sp,32    # Set stack pointer back to original value
jr    ra
. . .
```

The assembly listing generated by GCC performs a considerable amount of stack interactions – it is instructive to trace the code and keep track of the stack locations.

[47] Refer to RISC-V ABIs Specification (https://lists.riscv.org/g/tech-psabi/attachment/61/0/riscv-abi.pdf) section 8.5.6 for more information on the procedure linkage table.

	SP		SP + offset		ra		s0/fp		s0/fp plus offset		a0	a1	a2	a3	a4	a5
	Address	**Content**	**Address**	**Content**	**Address**	**Content**	**Address**			**Content**						
	0x3fffff2c(	1		0x02aaaaeeb1e	Inaccessible											
addi sp,sp,-32	0x3fffff2a(	0		0x02aaaaeeb1e	Inaccessible		0x2aaabb1430			Inaccessible						
sd ra,24(sp)	0x3fffff2a0		0x3fffff2b8	02aaaaeeb1e												
sd s0,16(SP)	0x3fffff2a0		0x3fffff2b(	0x2aaabb1430												
addi s0,sp,32	0x3fffff2a0		0x3fffff2c0				0x3fffff2c0									
li a5, 100	0x3fffff2a0						0x3fffff2c0									100
sw a5,-20(s0)	0x3fffff2a0						0x3fffff2c0	0x3fffff2ac		100						
li a5, 200	0x3fffff2a0						0x3fffff2c0									200
sw a5,-24(s0)	0x3fffff2a0						0x3fffff2c0	0x3fffff2a8		200						
lw a4,-24(s0)	0x3fffff2a0						0x3fffff2c0	0x3fffff2a8							200	
lw a5,-20(s0)	0x3fffff2a0						0x3fffff2c0	0x3fffff2ac							200	100
mv a1, a4	0x3fffff2a0						0x3fffff2c0					200			200	101
mv a0, a5	0x3fffff2a0						0x3fffff2c0				100	200			200	102
jal getproduct	0x3fffff2a0				0x100e4		0x3fffff2c0				100	200			200	103
mul a0, a0, a1	0x3fffff2a0		0x3fffff2a0		0x100e4		0x3fffff2c0				20000	200			200	104
ret	0x3fffff2a0				0x100e4		0x3fffff2c0				20000				200	105
mv a5, a0	0x3fffff2a0				0x100e4		0x3fffff2c0				20000				200	20000
sw a5,-28(s0)			0x				0x3fffff2c0	0x3fffff2a(		20000					Result in a5	

shows how a segment of the program interacts with the stack.

Table 7-1 Partial Stack interaction of listing7-1

	SP		SP + offset		ra		s0/fp		s0/fp plus offset		a0	a1	a2	a3	a4	a5
	Address	Content	Address	Content	Address	Content	Address			Content						
	0x3fffff2c(	1			0x02aaaaeeb1e	Inaccessible										
addi sp,sp,-32	0x3fffff2a(	0			0x02aaaaeeb1e	Inaccessible										
sd ra,24(sp)	0x3fffff2a0		0x3fffff2b:	02aaaaeeb1e			0x2aaabb1430			Inaccessible						
sd s0,16(SP)	0x3fffff2a0		0x3fffff2b	0x2aaabb1430												
addi s0,sp,32	0x3fffff2a0		0x3fffff2c0				0x3fffff2c0									
li a5, 100	0x3fffff2a0						0x3fffff2c0									100
sw a5,-20(s0)	0x3fffff2a0						0x3fffff2c0	0x3fffff2ac		100						
li a5, 200	0x3fffff2a0						0x3fffff2c0									200
sw a5,-24(s0)	0x3fffff2a0						0x3fffff2c0	0x3fffff2a8		200						
lw a4,-24(s0)	0x3fffff2a0						0x3fffff2c0	0x3fffff2a8							200	
lw a5,-20(s0)	0x3fffff2a0						0x3fffff2c0	0x3fffff2ac							200	100
mv a1, a4	0x3fffff2a0				0x100e4		0x3fffff2c0					200			200	101
mv a0, a5	0x3fffff2a0						0x3fffff2c0				100	200			200	102
jal getproduct	0x3fffff2a0				0x100e4		0x3fffff2c0				100	200			200	103
mul a0, a0, a1	0x3fffff2a0		0x3fffff2a0		0x100e4		0x3fffff2c0				20000	200			200	104
ret	0x3fffff2a0				0x100e4		0x3fffff2c0				20000				200	105
mv a5, a0	0x3fffff2a0				0x100e4		0x3fffff2c0				20000				200	20000
sw a5,-28(s0)			0x				0x3fffff2c0	0x3fffff2a(		20000				Result in a5		

7.1. Using in-line code

The next program uses in-line code to execute assembly language instructions from a single C source program . The GNU assembler keyword `asm` is used to denote the operands using C syntax. There are two forms of ASM– Basic and Extended. In-line assembly code is a bridge for interfacing the high level convenience of C/C++ to the low-level functionality of RISC-V assembly code

7.1.1. Basic ASM

Basic ASM is a set of assembly instructions. With inline code the `asm` keyword is not an actual C keyword[48] but it is understood by the assembler. Note that non-GNU assemblers may use an alternative keyword. Basic ASM is simpler than extended ASM and can be used when no operands are involved.

7.1.1. Extended ASM

Extended ASM can use variables from the C/C++ source code. Extended ASM cannot be used outside of these functions. The assembler template consists of:

```
asm(code template :  output operand(s) : input operand(s) : clobber list);
```

Table 7-2 gives an explanation.

Table 7-2 Inline assembly template

Template		
	Example	Description
Code - Assembler Instruction	`"mul %0, %1, %2"`	Regular assembly instruction
Code Template parameters	`add%[inputa], %[inputb]`	Using parameters passed as inputs to the code template
Output Operand(S) List	`: "=r" (result)` Can be left empty using :	List of output operand(s) [answer] is a symbolic name, r is a constraint string meaning register and (result) is returned to the Calling code.
Input Operands List	`[inputa] "r" (a), [inputb] "r" (b)`	Similar syntax to operand list
Clobber List	`“t0”, “t1”`	Optional list of registers, that *may* not be preserved

An example of in-line assembly code in a C program using Extended ASM is shown in Listing 7-2

[48] This is not the case with C++.

Listing 7-2 Extended ASM example

```
cat listing7-2.c

#include <stdio.h>

int main() {

int number1 = 15, number2 = 27, result;

// Using extended ASM to add a and b

asm volatile (

"add %0, %1, %2"

: "=r"(result)  // Output operand

: "r"(number1), "r"(number2)  // Input operands

:                // No clobbered registers

);

printf("Result of adding %d to %d is: %d\n", number1,number2,result);

return 0;

}}
```

- This instruction adds two registers indicated by the input operands %1 and %2,
- %0 represents the output operand.
- "=r" indicates that the result (sum) will be stored in a register.
- "r"(number1), "r"(number2) indicates registers.

The intermediate assembly file generated by the C compiler is shown below:

Note the comments are not generated by gcc but edited in for clarity.

```
.file   "listing7-2.c"

.option pic

.attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"

.attribute unaligned_access, 0
```

```
.attribute stack_align, 16
.text
.section        .rodata
.align  3
.LC0:
.string "Result of adding %d to %d is: %d\n"
.text
.align  1
.globl  main
.type   main, @function
main:
addi    sp,sp,-32
sd      ra,24(sp)
sd      s0,16(sp)
addi    s0,sp,32
li      a5,15     //number1 is stored in register a5
sw      a5,-20(s0) // It is then pushed onto the stack
li      a5,27     // number2 is stored in register a5
sw      a5,-24(s0) // It is then stored onto the next stack location
lw       a5,-20(s0) // number1 is retrieved from the stack and stored in
register a5
lw       a4,-24(s0) //number2 is retrieved from the stack and stored in
register a4
#APP
# 8 "listing7-2.c" 1
```

```
add a5, a5, a4 // Number1 is added to number2 storing result in register
a5
# 0 "" 2
#NO_APP
sw      a5,-28(s0) // Result is stored onto the stack
lw        a3,-28(s0) // Result is popped from the stack and loaded into
register a3
lw      a4,-24(s0) // Restore number2 into register a4
lw      a5,-20(s0) // Restore number1 into register a5
mv      a2,a4     // Store numbers into registers a1 and a2
mv      a1,a5
lla     a0,.LC0   // Set up print output
call    printf@plt
li      a5,0
mv      a0,a5
ld      ra,24(sp)
ld      s0,16(sp)
addi    sp,sp,32
jr      ra
.size   main, .-main
.ident  "GCC: (Debian 12.2.0-14) 12.2.0"
.section        .note.GNU-stack,"",@progbits
```

7.2. Optimizing code with GCC

Looking at the listing it appears that a deal of optimization could be performed. GCC features a number of optimization options[49]. The option -Os optimizes for code size and the intermediate assembly code is shown below

```
.file   "listing7-2.c"
.option pic
.attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
.attribute unaligned_access, 0
.attribute stack_align, 16
.text
.section        .rodata.str1.8,"aMS",@progbits,1
.align  3
.LC0:
.string "Result of adding %d to %d is: %d\n"
.section        .text.startup,"ax",@progbits
.align  1
.globl  main
.type   main, @function
main:
addi    sp,sp,-16
sd      ra,8(sp)
li      a3,15
li      a5,27
#APP
# 8 "listing7-2.c" 1
```

[49] Default is no optimization.

```
add a3, a3, a5

# 0 "" 2

#NO_APP

li      a2,27

sext.w  a3,a3

li      a1,15

lla     a0,.LC0

call    printf@plt

ld      ra,8(sp)

li      a0,0

addi    sp,sp,16

jr      ra

.size   main, .-main

.ident  "GCC: (Debian 12.2.0-14) 12.2.0"

.section        .note.GNU-stack,"",@progbits
```

The optimization options are shown in Table 7-3.

Table 7-3 GCC optimization options

Optimization Option	Description
-O0	Default, reduces compilation time. Easier to debug.
-O1	Good compilation time, attempts to reduce code size and execution time.
-O2	Performs more optimizations not involving a tradeoff between size and speed.
-O3	Maximum optimization, can increase compilation time.
-Os	Optimize for size, includes most of the O2 optimization unless they incur increased code size.
-Ofast	Enables O3 optimizations and optimizations at the expense of standards compliance.
-Og	Good compromise for debugging with a good degree of optimization.
-Oz	Aggressive optimization for size.

The `clang` compiler uses similar optimization options. A comparison of the assembly file size using the `wc` utility[50] is shown following:

```
$ clang -O0 -S  listing7-2.c && wc listing7-2.s

52  182 1463 listing7-2.s

$ clang -O1 -S  listing7-2.c && wc listing7-2.s

39  124 1015 listing7-2.s

$ clang -O2 -S  listing7-2.c && wc listing7-2.s

39  124 1015 listing7-2.s

$ clang -O3 -S  listing7-2.c && wc listing7-2.s

39  124 1015 listing7-2.s

$ clang -Os -S  listing7-2.c && wc listing7-2.s

39  124 1015 listing7-2.s

$ clang -Ofast -S  listing7-2.c && wc listing7-2.s

39  124 1015 listing7-2.s

$ clang -Og -S  listing7-2.c && wc listing7-2.s

39  124 1015 listing7-2.s

$ clang -Oz -S  listing7-2.c && wc listing7-2.s

39  124 1015 listing7-2.s
```

The next listing shows an example of in-line Basic ASM used with C code; in practice Extended ASM is used more often with in-line assembly code.

Listing 7-3 Basic ASM example

```
# include <stdio.h>

const char message[] = "Hello - RiscV Basic ASM!\n";

int main()

{
```

[50] `wc` counts lines, words and bytes so an output of 52 182 1463 refers to 52 lines, 182 words and 1463 bytes; using `wc -l` will only return the line count

```
asm(
"la a0, message\n"       // load address of msg into a0 (1st argument to
puts)
"call puts\n"           // call puts(message)
"li a7, 93\n"
"ecall\n"
);
return 0;
}
```

7.3. Format Specifiers

Earlier programs in this chapter have already used `printf` to output results. The C standard library function `printf` is defined within `<stdio.h>` as `int printf(const char *format,…)` It is a *variadic* function which means that it can take a variable number of arguments. This is conveyed by the ellipsis… in the prototype. The function takes a minimum of one argument which is a pointer to the location of the starting character of the text. The text itself can embed formatting tags which specify how the arguments that are passed are to be printed – for example a variable using "%d" will be formatted as a signed base 10 integer.

A non-exhaustive list of format specifiers is shown in Table 7-4.

Table 7-4 printf format specifiers

Format specifier	interpretation
%d	Signed decimal number.
%u	Unsigned decimal number.
%s	Pointer to an array of characters.
%c	Outputs a single character.
%x	Represents an unsigned integer in lower case hexadecimal form.
%X	Represents an unsigned integer in upper case hexadecimal form.
%%	Outputs a literal "%" character.
%e	Represents floating point as decimal exponent notation.
%f	Represents floating point as decimal.

Using the `printf` specifiers help immensely when using assembly code, although it should be noted that some systems will not have `printf` available[51]. Listing 7-3 shows examples. Here the registers a0 through a3 are used as function parameters to printf as specified in the ABI calling convention. The use of `printf` could conceivably slow down execution in time-dependent code whereas the direct assembly printing methods are faster. The `printf` function expects the first parameter to be a null-terminated string, which uses the `.asciz` assembler directive.

Listing 7-4 Using the printf function with assembly code

```
$ cat listing7-4.s

.text

.global _start

_start:

la a0, string1

li a1, -45

li a2, 88

add a3,a1,a2

call printf


la a0, string2

li a1, 55

mul a2, a1, a1

call printf

li a0, 0

call exit

.data
```

[51] Typically this would apply to bare-metal embedded implementations with limited resources.

```
string1: .asciz "The first number (-45) represented as signed decimal is
%d, the second number (88) represented as signed decimal is %d, the addition
of the two numbers represented as lower case hexadecimal is %x\n"
string2: .asciz "The decimal number %d squared in upper case hexadecimal
form is %X\n"
```

Output:

```
$ ./listing7-4
The first number (-45) represented as signed decimal is -45, the second
number (88) represented as signed decimal is 88, the addition of the two
numbers represented as lower case hexadecimal is 2b
The decimal number 55 squared in upper case hexadecimal form is BD1.
```

Exercises for chapter 7

1. Write a program that could benefit from optimization; include redundant instructions to see how it is handled by the disassembled code
2. Generate compute intensive code and see if optimization can reduce run-time.

Summary of RISC-V instruction used in chapter 7

Function Calling and Stack Use

sd – Store doubleword (save registers to stack)

ld – Load doubleword (restore registers from stack)

jal – Jump and link (function calls)

ret – Return from function (pseudo-instruction for jalr)

Macro and Inline ASM Tools

- While these are not actual RISC-V instructions, the chapter discusses **basic and extended inline assembly syntax** in GCC using:
    - asm("...") or __asm__ volatile ("...") blocks
    - **Constraints** like r, m, =r, 0, etc.
    - Clobber lists and output/input operands

Chapter 8. Floating-Point

Overview of the chapter

Chapter 8 introduces floating-point operations in RISC-V, based on the IEEE 754 standard. It explains how floating-point numbers are represented, manipulated, and evaluated in assembly, highlighting both single and double precision.

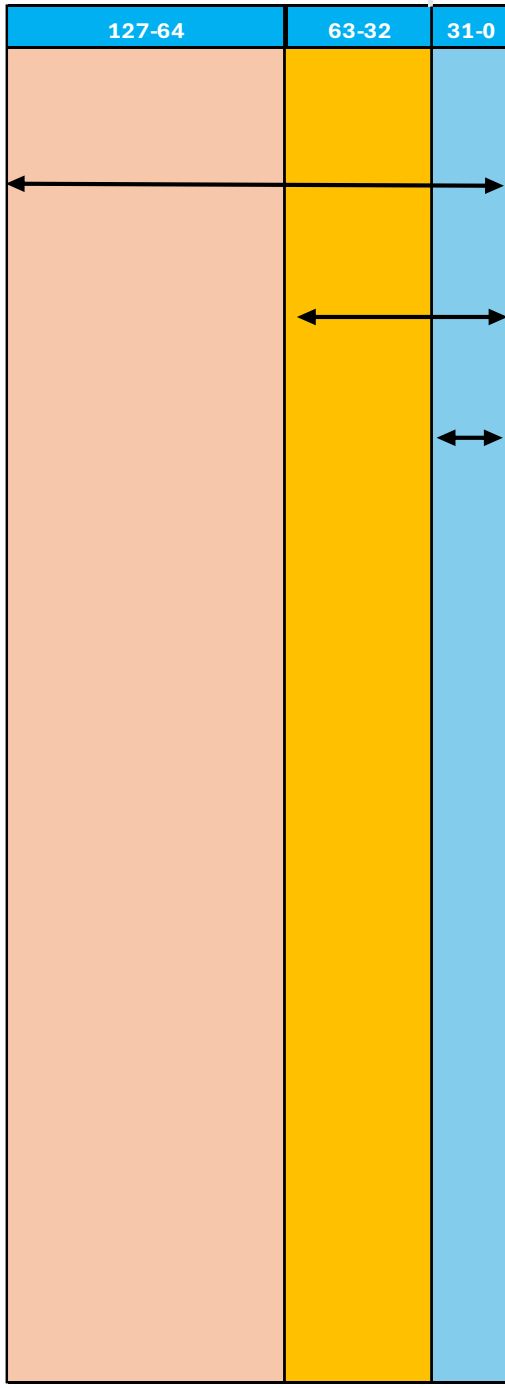
8.1. RISC-V floating-point capability

Not all RISC-V systems can handle floating point; recall that only some RISC-V systems have floating-point support as evidenced from their identification string (the F extension), as discussed on page 2-2.

8.1.1. Floating-point register set

Capable systems have 32 (f0 → f31) floating-point registers shown in Figure 8-1, while the register width (FLEN) is determined by the RV extension shown in Table 8-2.

Figure 8-1 Floating-point registers



Floating-Point registers		
Register Name	ABI Name	Saver responsibity
f0	ft0	Caller
f1	ft1	Caller
f2	ft2	Caller
f3	ft3	Caller
f4	ft4	Caller
f5	ft5	Caller
f6	ft6	Caller
f7	ft7	Caller
f8	fs0	Callee
f9	fs1	Callee
f10	fa0	Caller
f11	fa1	Caller
f12	fa2	Caller
f13	fa3	Caller
f14	fa4	Caller
f15	fa5	Caller
f16	fa6	Caller
f17	fa7	Caller
f18	fs2	Callee
f19	fs3	Callee
f20	fs4	Callee
f21	fs5	Callee
f22	fs6	Callee
f23	fs7	Callee
f24	fs8	Callee
f25	fs9	Callee
f26	fs10	Callee
f27	fs11	Callee
f28	ft8	Caller
f29	ft9	Caller
f30	ft10	Caller
f31	ft11	Caller

32 floating -point registers, data width is determined by RV extension

Table 8-2 indicates the number of bits that are used by floating-point numbers in the RISC-V architecture, for reference a single precision number is referenced as a *float* in the C language and a double precision number as a *double*.

This chapter will only discuss single and double precision numbers not half or quad.

Recall from chapter one:

- Single precision numbers are divided into three fields with a single sign bit, eight bits for a biased exponent and 23 bits for the significand.
- Double precision numbers are divided into three fields with a single sign bit, eleven bits for a biased exponent and 52 bits for the significand.
- With normalized numbers the leading 1.XXX… is implicit and not coded.

Table 8-1 Bit fields of single and double precision floating-point numbers

Format	Bits	Significand	Unbiased Exponent	Decimal Precision
Single	32	24 (23+1)	8	6-9 digits
Double	64	53 (52+1)	11	15-17 digits

Table 8-2 Floating-point register width

Optional extension	Register width
H Half precision	16 bits (FLEN)
F Single precision	32 bits (FLEN)
D Double precision	64 bits (FLEN)
Q Quad precision	128 bits (FLEN)

8.2. Instruction types

Floating-point instructions can be broadly categorized into the following areas.

8.2.1. Arithmetic instructions

Floating-point arithmetic operations include –

- Add
- Subtract
- Multiply
- Divide
- Square root
- Minimum
- Maximum

8.2.2. Load and store instructions

- Load
- Store

8.2.3. Convert instructions

- Convert from float to unsigned integer
- Convert from unsigned integer to float
- Convert from single precision float to double precision float
- Convert from double precision float to single precision float

8.2.4. Categorization instructions

These are used to ascertain the type of value such as minus infinity -∞, -0, NaN, . . . The `fclass` instructions are used to store a value corresponding to the type in a destination.

8.2.5. Comparison instructions

This covers the usual comparisons – less than or equal, equal,

8.2.6. Miscellaneous instructions

- Sign-injection which copies from a source to a destination with sign-bit manipulation.

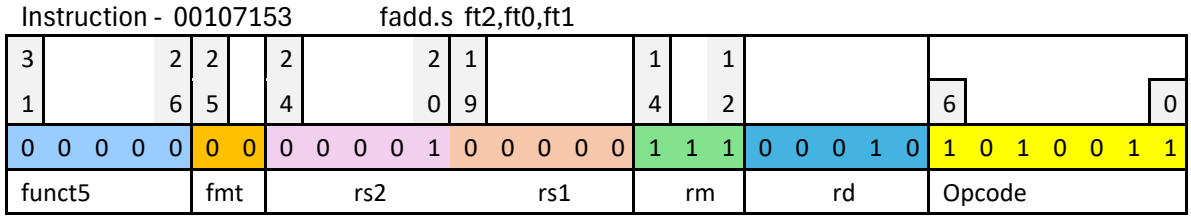
8.3. Instruction format

The format of a floating-point instruction is `F<instruction>.<precision> rd, rs1, rs2` where <instruction> is an operation such as <`ADD`>, <`MUL`> or <`DIV`> and <precision> is the floating point precision such as <S> or <D>, so the instruction `FSUB.S` refers to a single precision floating point subtraction operation. An arithmetic instruction – `FADD.S f0,f1,f2` will add the contents of registers f1 and f2 placing the result in register f0. The field breakdown of this instruction is as follows:

Table 8-3 Field meaning of FADD.s instruction

Field	Value	Notes
Opcode	1010011 (53)	Used with funct5 to determine operation
rd	00010 (2)	Destination register (F2)
rm	111 (7)	Select the dynamic rounding mode held in frm (rounding mode field) which default mode if not specified in the instruction
rs1	00000	First source register (F2)
rs2	00001	Second source register (F1)
fmt	00	S (32-bit) Single precision see
funct5	00000	FADD instruction

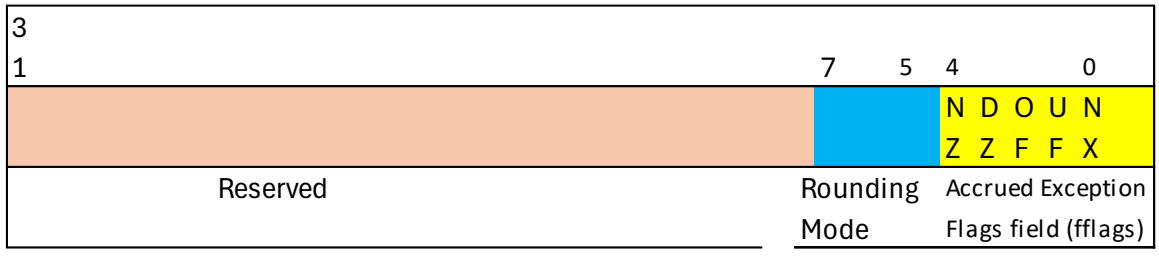
Figure 8-2 FADD bit fields



8.3.1. Floating point control and status register

The floating-point control and status register (**FCSR**) is a 32-bit register that is used to flag exceptions and the *rounding mode* that is used with floating point operations. The exception flags occupy bits 4:0 and the rounding mode occupies bits 7:5 as shown in Figure 8-3.

*Figure 8-3 **FCSR** bit definitions*

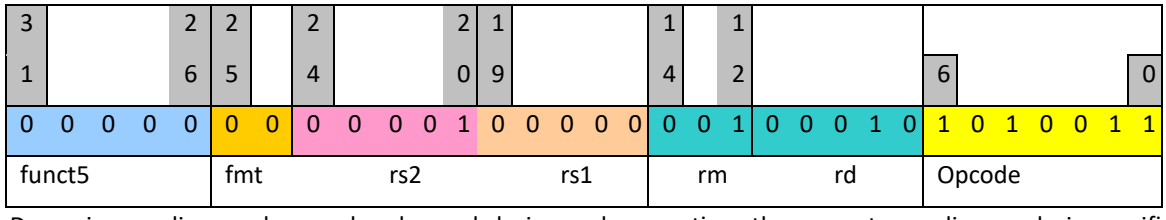


8.3.2. Rounding Modes

There are two types of rounding modes - *dynamic* and *static*. Static rounding modes are specified in the floating-point instruction such as `fadd.s ft2, ft0. ft1, rtz` where `rtz` stands for *round towards zero*. Static rounding modes are fixed.

The bit field breakdown for `fadd.s ft2, ft0. ft1, rtz` is shown in Figure 8-4

Figure 8-4 Field breakdown of FADD.s f2,f0,f, rtz instruction



Dynamic rounding modes can be changed during code execution; the current rounding mode is specified within the **fcsr** register.

Table 8-4 defines the various encoding modes:

Table 8-4 Rounding mode bits

Mode	Mnemonic	Notes
000	RNE	Round to nearest (even values are preferred)

001	RTZ	Round towards zero
010	RDN	Round down towards -infinity
011	RUP	Round up towards +infinity
100	RMM	Round to nearest (max magnitude)
101	Reserved	
110	Reserved	
111	DYN	Dynamic rounding

8.3.3. Accrued Exception bits

The meaning of the exception bits are:

- NZ Invalid
- OF Overflow
- UF Underflow
- DZ Divide by zero
- NX Inexact

The `frcr rd` command can be used to read the register placing the result into a general-purpose (integer) register and the `fscsr rs1` instruction is used to set bits from a source general-purpose register.

Note that the accrued exception bits must be cleared by the software once they have been set!

The first listing in this section adds two double precision floating-point numbers and uses printf[52] to print the result. The address of the numbers pi and e are first placed in the integer registers a0 and a1. They are then loaded into floating-point registers fa0 and fa1. They are added together, placing the result in fa2 by the `fadd.d` instruction. After this the floating-point values are placed back into the integer registers so that they can be displayed using `printf`.

Listing 8-1 Adding two double-precision floating-point numbers

```
# Double-precision floating-point addition example

.data

pi:    .double 3.141592653589793    # First double-precision number

euler:    .double 2.718281828459045    # Second double-precision number
```

[52] Use double-precision with printf. See assembly - How to print a single-precision float with printf - Stack Overflow (https://stackoverflow.com/questions/37082784/how-to-print-a-single-precision-float-with-printf) for elaboration.

```
displayresult: .string "Pi %.15f added to e %.15f = %.15f\n"   # Format
string for printf
.text
.global main
main:
        # Load double-precision floating-point numbers
        la a0, pi             # Load address of pi
        fld fa0, 0(a0)          # Load pi value into fa0
        la a0, euler             # Load address of e
        fld fa1, 0(a0)          # Load e value into fa1
        fadd.d fa2, fa0, fa1   # Double-precision addition, result in fa2
        # set printf arguments
        # Move Floating-point numbers into integer registers
        fmv.x.d a1, fa0         # pi goes to a1
        fmv.x.d a2, fa1         # e goes to a2
        fmv.x.d a3, fa2         # Result to a3
        # Load string
        la a0, displayresult   # printf a0 for string, a1,a2,... for other
parameters
        # Print the result
        call printf
        # exit
        li a7, 93
        ecall
```

The compilation string used was:

```
$ gcc -g -listing8-1.s -o listing8-1
```

and the output shows:

```
 ./listing8-1

Pi 3.141592653589793 added to e 2.718281828459045 = 5.859874482048838
```

After the floating-point registers have been added their contents shows:

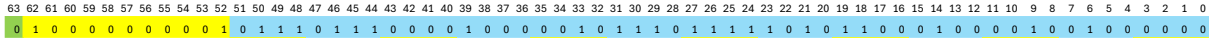
```
fa0            {float = 3.37028055e+12, double = 3.1415926535897931}    (raw 0x400921fb54442d18)
fa1            {float = -2.85695233e-32, double = 2.7182818284590451}   (raw 0x4005bf0a8b145769)
fa2            {float = -1.06623026e+29, double = 5.8598744820488378}   (raw 0x40177082efac4240)
```

After the floating-point values have been moved back into the integer registers, their contents are:

```
a0              0x2aaaaac010      183251943440
a1              0x400921fb54442d18        4614256656552045848
a2              0x4005bf0a8b145769        4613303445314885481
a3              0x40177082efac4240        4618283650560836160
```

To verify:

Convert 40177082efac4240 to binary



Extract sign bit (bit 63) = 0 = Positive

Extract Exponent field (bits 62:52) = 1025 decimal, double precision range is -1022→ +1023, biased exponent is 1025-1023 = 2

Extract significand field bits 51:0), adding explicit leading 1 to get

1.0111011100001000001011101111101011000100001001000000

$= (1 \times 2^{0}) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) + (1 \times 2^{-4}) + (0 \times 2^{-5}) + (1 \times 2^{-6}) + (1 \times 2^{-7}) + (1 \times 2^{-8}) + (0 \times 2^{-9})$

4. $+ (0 \times 2^{-10}) + (0 \times 2^{-11}) + (0 \times 2^{-12}) + (1 \times 2^{-13}) + (0 \times 2^{-14}) + (0 \times 2^{-15}) + (0 \times 2^{-16}) + (0 \times 2^{-17}) + (0 \times 2^{-18}) + (1 \times 2^{-19})$

5. $+ (0 \times 2^{-20}) + (1 \times 2^{-21}) + (1 \times 2^{-22}) + (1 \times 2^{-23}) + (0 \times 2^{-24}) + (1 \times 2^{-25}) + (1 \times 2^{-26}) + (1 \times 2^{-27}) + (1 \times 2^{-28}) + (1 \times 2^{-29}) + (0 \times 2^{-30}) + (1 \times 2^{-31}) + (0 \times 2^{-32}) + (1 \times 2^{-33}) + (1 \times 2^{-34}) + (0 \times 2^{-35}) + (0 \times 2^{-36}) + (0 \times 2^{-37}) + (1 \times 2^{-38}) + (0 \times 2^{-39}) + (0 \times 2^{-40}) + (0 \times 2^{-41}) + (0 \times 2^{-42}) + (1 \times 2^{-43}) + (0 \times 2^{-44}) + (0 \times 2^{-45}) + (1 \times 2^{-46}) + (0 \times 2^{-47}) + (0 \times 2^{-48}) + (0 \times 2^{-49}) + (0 \times 2^{-50}) + (0 \times 2^{-51}) + (0 \times 2^{-52})$

6. $= (1.46496862051220944068)_{10}$

Multiply by exponent (obtained earlier) = $1.46496862051220944068 * (1*2^{2}) = \sim 5.86$

The next section of code introduces the floating-point multiply and divide instructions along with integer conversion with different types of rounding. Two numbers are multiplied together and then this result is divided by one of the original numbers to see if there are any errors due to precision.

Listing 8-2 Floating-point rounding using static modes

```
.data
```

```
number1:                    .double 123.141592653589793    # First double-
precision number
number2:                    .double 422.718281828459045    # Second double-
precision number
displaymresult:          .asciz "\n The result of %.15f multiplied by %.15f
= %.15f\n"  # Format string for printf
displaydresult:          .asciz "\n The result of %.15f divided by %.15f =
%.15f\n"
displayrne:               .asciz "\n The integer result rounded to nearest
(ties to even) is %d\n"
displayrup:              .asciz "\n The integer result rounded up is %d\n"
displayrdn:             .asciz "\n The integer result rounded down is %d\n"
displayrmm:               .asciz "\n The integer result rounded to nearest
(max magnitude) is %d\n"
.text
.global main
main:
   # Load double-precision floating-point numbers
   la a0, number1          # Load address of first number
   fld fa0, 0(a0)          # Load number1 value into fa0
   la a0, number2          # Load address of second number
    fld fa1, 0(a0)          # Load number2 value into fa1
    fmul.d fa2, fa0, fa1   # Double-precision multiplication, result in fa2
    fdiv.d fa3, fa2, fa0   # Now divide (number1*number2)by number1 result
in fa3
        fcvt.lu.d t0,fa2,rne
```

```
        fcvt.lu.d t1,fa2,rup
        fcvt.lu.d t2,fa2,rdn
        fcvt.lu.d t3,fa2,rmm
        # Set up stack space and push registers t0-t3
        addi sp,sp,-48 #Allocate space
        sd t0, 8(sp)
        sd t1,16(sp)
        sd t2,24(sp)
        sd t3,32(sp)
        # set printf arguments for fa2 value
        # Move Floating-point numbers into integer registers
        fmv.x.d a1, fa0        # number1 goes to a1
        fmv.x.d a2, fa1        # number2 goes to a2
        fmv.x.d a3, fa2        # Result to a3
        # Load multiplication string
        la a0, displaymresult   # printf a0 for string, a1,a2,... for other
parameters
        call printf
        # Print the division result
        la a0, number1
        fld fa0, 0(a0)
        fmv.x.d a1, fa2 #  multiplication result goes into parameter1
        fmv.x.d a2, fa0 # Number1 is parameter2
        fmv.x.d a3, fa3 # derived number2 is parameter3


        # Load division string
```

```
        la a0, displaydresult
        call printf
        # Now show rounding values and pop stack values
        ld a1, 8(sp) # Pop t0 to a0
        la a0, displayrne
        call printf
        ld a1, 16(sp) # Now pop t1 onto a1
        la a0, displayrup
        call printf
        ld a1, 24(sp) # Pop t2
        la a0, displayrdn
        call printf
        ld a1, 32(sp) # Pop t3
        la a0, displayrmm
        call printf
        # Restore stack pointer
        addi sp,sp,48
        # exit
        li a7, 93
        ecall
```

Output:

```
$ ./listing8-2
 The result of 123.141592653589797 multiplied by 422.718281828459055 =
52054.202468145471357
```

```
 The result of 52054.202468145471357 divided by 123.141592653589797 =
422.718281828459055
 The integer result rounded to nearest (ties to even) is 52054
 The integer result rounded up is 52055
 The integer result rounded down is 52054
 The integer result rounded to nearest (max magnitude) is 52054
```

Listing 8-3 Using dynamic rounding mode

```
# listing 8-3, use of dynamic rounding
.section .data
        pi:         .float 3.141
        formatrup:      .asciz "\n Pi Rounded up result (RUP) is %d\n"
        formatdn:       .asciz "\n Pi Rounded down result (RDN) is %d\n"
        .equ    roundingmask, 0xe0
        .equ    rup,    0x60
        .equ    rdn, 0x40
.section .text
        .global main
        .extern printf
main:
    # Set the rounding mode to round up (0x60)
    frcsr t0                  # Read FCSR into t0
    li    t1, roundingmask    # Mask to clear rounding bits
    not   t1, t1
    and   t0, t0, t1          # Clear bits 7:5
    li    t2, rup
```

```
    or    t0, t0, t2
    fscsr t0                  # Write updated FCSR to t0
    # Load Pi into f0
    la    t3, pi
    flw   f0, 0(t3)
    # Convert to int using dynamic rounding mode (RUP)
    fcvt.w.s a1, f0           # Result goes to a1 (first int argument)
    # Load format string into a0 (first arg for printf)
    la    a0, formatrup
    call  printf
        # Do it again, this time round down
        frcsr t0                  # Read FCSR into t0
        li    t1, roundingmask            # Mask to clear rounding bits
        not   t1, t1
        and   t0, t0, t1          # Clear bits 7:5
        li    t2, rdn            #  0x40 (RDN)
        or    t0, t0, t2
        fscsr t0                  # Write updated FCSR
        # Load Pi into f0
        la    t3, pi
        flw   f0, 0(t3)
        # Convert to int using dynamic rounding mode (RDN)
        fcvt.w.s a1, f0           # Result goes to a1 (first int argument)
        # Load format string into a0 (first arg for printf)
        la    a0, formatdn
        call  printf
```

```
        # Return 0 from main
        #li    a0, 0
        #ret

        li a7, 93
ecall
```

Output

```
$ ./listing8-3

 Pi Rounded up result (RUP) is 4
 Pi Rounded down result (RDN) is 3
```

Before looking at floating-point compare instructions Listing 8-4 generates the square root of two numbers. The first number (2) does not have an exact[53] square root whereas the second number (9) does.

Listing 8-4 Use of sqrt instruction and reading the FCSR register

```
# Listing 8-4.s square root function and reading the fcsr register
# This code could be improved on greatly by showing the actual instruction
that flagged the condition
.section .data
message1:     .asciz "\n The square root of the number 9 and 2 when squared
is approximately %.14f and %.14f\n"
fcsrerrormsg:     .asciz "\n Warning fcsr flags set; the hex value read is
%d\n"
accexceptbitsmsg: .asciz "\n 1 = NX (Inexact)\n 2 = UF (Underflow)\n 4 =
OF (Overflow)\n 8 = DZ (Divide by zero)\n 10 = NV (Invalid)\n"
square1:    .double 9.0
```

[53] The square root of 2 is *irrational*

```
square2:    .double 2.0
.equ flagmask, 0x1f


.section .text


.global main
.extern printf
main:
      la t0, square1
      la t1, square2
      fld fa0, 0(t0)
      fld fa1, 0(t1)
      fsqrt.d fa2, fa0
      fsqrt.d fa3, fa1
      fmul.d fa4, fa2, fa2
      fmul.d fa5, fa3, fa3
      la a0, message1
      fmv.x.d a1,fa4
      fmv.x.d a2,fa5
      call printf
      li t2, flagmask # Not interested in the rounding bits this time
        frcsr t0        # read fcsr register
        and t0, t0,t2
        beq t0, x0, exit
        la a0, fcsrerrormsg
        mv a1, t0
```

```
      call printf

      la a0, accexceptbitsmsg

      call printf

exit:

      li a7, 93

      ecall
```

Output

```
$ ./listing8-4

 The square root of the number 9 and 2 when squared is approximately

9.00000000000000 and 2.00000000000000

 Warning fcsr flags set; the hex value read is 1

 1 = NX (Inexact)

 2 = UF (Underflow)

 4 = OF (Overflow)

 8 = DZ (Divide by zero)

 10 = NV (Invalid)
```

Note that after the instruction fsqrt.d fa2, fa0 (square root of 9) has been executed the FCSR register looks like:

```
 24          fsqrt.d fa2, fa0
 25          fsqrt.d fa3, fa1
 26          fmul.d fa4, fa2, fa2
 27          fmul.d fa5, fa3, fa3
 28          la a0, message1
 29          fmv.x.d a1,fa4
 30          fmv.x.d a2,fa5
 31          call printf
 32
 33          li t2, flagmask # Not interested in the rounding bits
 34          frcsr t0        # read fcsr register
 35          and t0, t0,t2
```

```
thre Thread 0x3ff7fc3c60 (regs) In: main
g symbols from listing8-4...
b 1
oint 1 at 0x714: file listing8-4.s, line 20.
run
ng program: /home/alan/asm/chapter08/listing8-4
d debugging using libthread_db enabled]
host libthread_db library "/lib/riscv64-linux-gnu/libthread_db.so.1".

oint 1, main () at listing8-4.s:20
n
n
i reg fcsr
        0x0       NV:0 DZ:0 OF:0 UF:0 NX:0 FRM:0 [RNE (round to nearest; ties to even)]
```

No fcsr bits set after the square root of 9 has been calculated

After the instruction `fsqrt.d fa3, fa1` (square root of 2) has completed GDB shows that the Inexact bit has been set. This will normally indicate that rounding had to be invoked.

```
  >     26          fmul.d fa4, fa2, fa2
        27          fmul.d fa5, fa3, fa3
        28          la a0, message1
        29          fmv.x.d a1,fa4
        30          fmv.x.d a2,fa5
        31          call printf
        32
        33          li t2, flagmask # Not interested in the rounding bits
        34          frcsr t0        # read fcsr register
        35          and t0, t0,t2
```

```
multi-thre Thread 0x3ff7fc3c60 (regs) In: main
Reading symbols from listing8-4...
(gdb) b 1
Breakpoint 1 at 0x714: file listing8-4.s, line 20.
(gdb) run
Starting program: /home/alan/asm/chapter08/listing8-4
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/riscv64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at listing8-4.s:20
(gdb) n
(gdb) n
(gdb) i reg fcsr
fcsr           0x0      NV:0 DZ:0 OF:0 UF:0 NX:0 FRM:0 [RNE (round to nearest; ties to even)]
(gdb) n
(gdb) i reg fcsr
fcsr           0x1      NV:0 DZ:0 OF:0 UF:0 NX:1 FRM:0 [RNE (round to nearest; ties to even)]
(gdb)
```

fcsr bit now set after the square root of 2 has been calculated

Care must be taken when making comparisons between floating-point numbers. After a flag has been set in the FCSR register, it is important to note that it must be cleared implicitly by the code. Usually, it is not necessary to check the state of the FCSR register after each floating-point computation has been executed as the boundaries are usually finite and known in advance.

8.4. Floating-Point comparison instructions

The floating-point comparison instructions are shown in Table 8-5.

Table 8-5 Floating-point comparison instructions

Instruction	Example	Explanation
feq.s|d[54]	`feq.d rd, rs1,rs2`	Write the value 1 to the integer register rd, if the double precision number in rs1 is equal to the double precision number in rs2, else write the value 0 to the integer register rd.
flt.s|d	`flt.s rd, rs1, rs2`	Write the value 1 to the integer register rd, if the single precision number in rs1 is less than the single precision number in rs2, else write the value 0 to the integer register rd.
fle.s|d	`fle.d rd, rs1, rs2`	Write the value 1 to the integer register rd, if the double precision number in rs1 is less than or equal to the double precision number in rs2, else write the value 0 to the integer register rd.

8.5. Floating-point classification instructions

The classify instructions are used to signify the properties of a floating-point number. There are ten bits available to specify a number's class only one of these bits set at any given time. Some of these classifications require further explanation as they were not discussed in chapter one –

- Subnormal A *subnormal* number or a *denormalized* number is a number that is smaller than can be expressed in normal format (1.000…) as described in the IEEE 754 standard. Subnormal numbers are closer to zero than can be expressed in normal format and have less precision.
- Signaling NaN An exception can be raised when NaN is encountered.
- Quiet NaN A quiet NaN does not signal an exception.

Table 8-6 lists the classification bits and their definitions.

Table 8-6 Floating-point classes

Bit	Interpretation when set
0 (1)	7. Negative infinity -∞
1 (2)	Negative normal

[54] Here "|" means or so the instruction could be `feq`.s or `feq.d`

2 (4)	Negative subnormal
3 (8)	Negative zero -0
4 (10)	Positive zero +0
5 (20)	Positive subnormal
6 (40)	Positive normal
7 (80)	Positive infinity +∞
8 (100)	Signaling NaN
9 (200)	Quiet NaN

The next program generates two classes of number – Subnormal and a quiet NaN. The annotated stages to generate the subnormal number are shown in Figure 8-6.

Listing 8-5 Classification of numbers - subnormal and quiet NaN

```
.section .data

minusone:       .double -1

.section .text

.global _start

_start:

    # Generate a subnormal number by applying division to two normal number

    # For RV64D systems (64-bit registers)

    # First Generate a 64-bit tiny number across t0 and t1

    li t0, 0x00100000   # Upper 32 bits of smallest normal double (2^-1022)

    li t1, 0x00000000      # Lower 32 bits

    # Set up divisor

    li t2, 0x40000000      # Upper 32 bits of 2.0

    li t3, 0x00000000      # Lower 32 bits

    # Consolidate into 64-bit values

    slli t0, t0, 32

    or t0, t0, t1          # t0 = 2^-1022 (smallest normal double)

    slli t2, t2, 32
```

```
        or t2, t2, t3           # t2 = 2.0

        # t0 and t2 now have full 64 bit values

        # Store them over to Floating-point registers

        fmv.d.x f0, t0          # f0 = 2^-1022

        fmv.d.x f1, t2          # f1 = 2.0

        # Divide them - produces 2^-1023 (subnormal)

        fdiv.d f2, f0, f1       # f2 = (2^-1022)/2 = 2^-1023

        # Verify the result is subnormal (exp=0, mantissa≠0)

        fmv.x.d t4, f2          # Get bit pattern

        fclass.d t0, f2

        # Expected result: 0x0008000000000000

        # Exponent bits (62:52) = 0

        # Mantissa bits (51:0) = 0x800000000000

        la t5, minusone

        fld f4, 0(t5)

        fsqrt.d f4, f4  # Square root of -1?

        fclass.d t1, f4

         # Exit

        li a7, 93               # Exit syscall number

        ecall
```

GDB shows the classification of the f2 and f4 registers after computation. Register f2 holds a value smaller than can be represented by normal numbers and is therefore classified as subnormal. Register f4 was used to calculate the square root of minus one which is a complex number and is categorized NaN.

Figure 8-5 GDB showing floating-point number classification

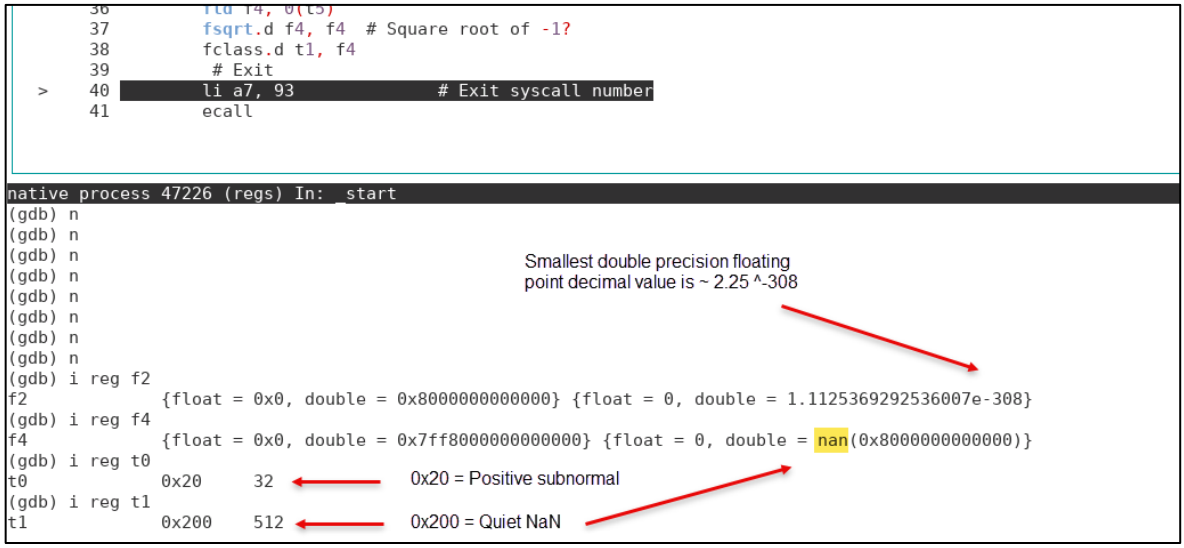
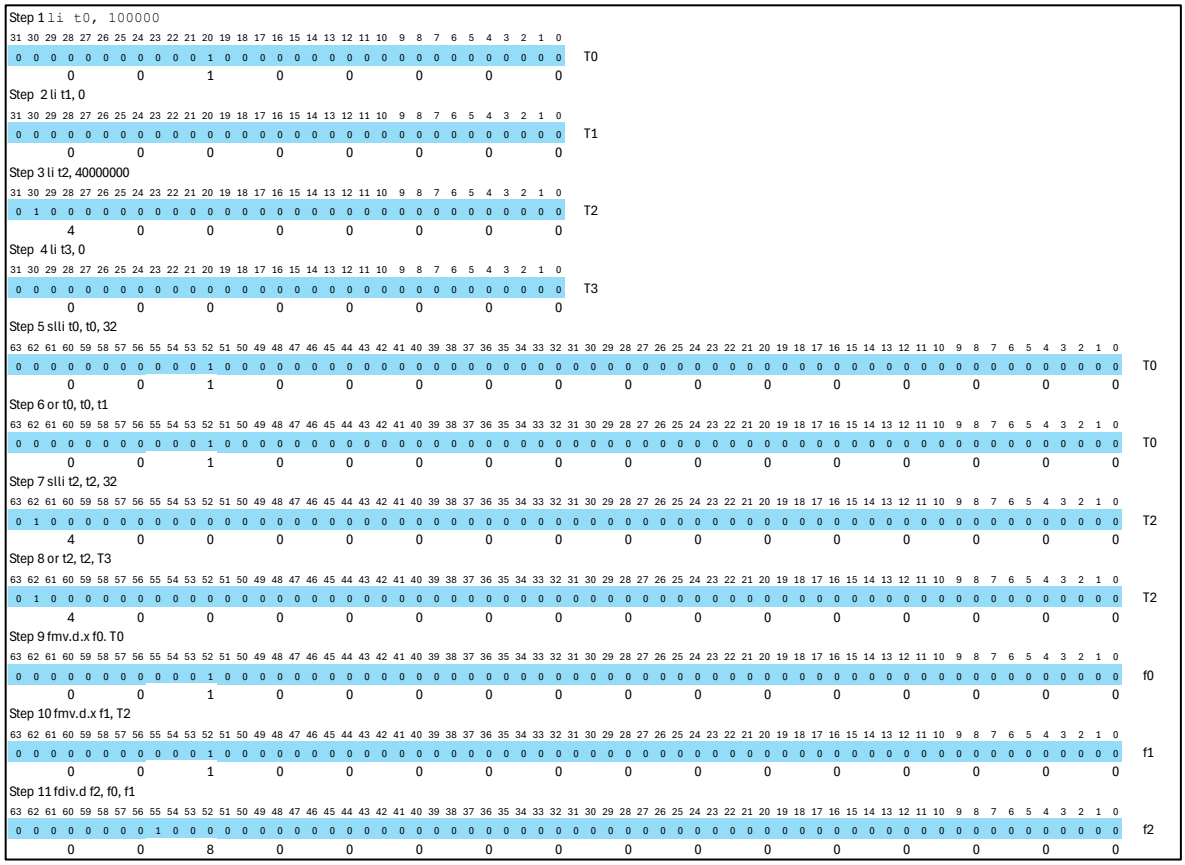

36 fld f4, 0(t5)
37 fsqrt.d f4, f4 # Square root of -1?
38 fclass.d t1, f4
39 # Exit
40 li a7, 93 # Exit syscall number
41 ecall
native process 47226 (regs) In: _start
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) i reg f2
f2 {float = 0x0, double = 0x8000000000000} {float = 0, double = 1.1125369292536007e-308}
(gdb) i reg f4
f4 {float = 0x0, double = 0x7ff8000000000000} {float = 0, double = nan(0x8000000000000)}
(gdb) i reg t0
t0 0x20 32
(gdb) i reg t1
t1 0x200 512
Smallest double precision floating point decimal value is ~ 2.25 ^-308
0x20 = Positive subnormal
0x200 = Quiet NaN

Figure 8-6 Annotated instruction steps to generate a subnormal number



8.6. Exercises for chapter 8

1. Write a program to generate different classes of floating-point numbers, print out the class of number that was produced.
2. Explain Bias as described in IEEE 754

8.7. Summary of RISC-V instructions used in chapter 8

Floating-Point Arithmetic Instructions

FADD.S – Floating-point add (single precision)

FSUB.S – Floating-point subtract (single precision)

FMUL.S – Floating-point multiply (single precision)

FDIV.S – Floating-point divide (single precision)

FSQRT.S – Square root (single precision)

FMIN.S / FMAX.S – Minimum / maximum (single precision)

(There are .D variants for double precision, e.g., FADD.D, FSUB.D.

Floating-Point Load/Store Instructions

FLW – Load single-precision float

FLD – Load double-precision float

FSW – Store single-precision float

FSD – Store double-precision float

Conversion Instructions

FCVT.W.S / FCVT.S.W – Convert between float and integer (single)

FCVT.WU.S / FCVT.S.WU – Convert unsigned integer ↔ float

FCVT.D.S / FCVT.S.D – Convert between single and double precision

---

Classification Instructions

FCLASS.S / FCLASS.D – Classify a floating-point value (Identifies if a value is NaN, infinity, subnormal, etc.)

Comparison Instructions

FEQ.S / FEQ.D – Compare for equality

FLT.S / FLT.D – Compare less than

FLE.S / FLE.D – Compare less than or equal

Miscellaneous Instructions

FSGNJ.S / FSGNJN.S / FSGNJX.S – Sign manipulation (sign-inject, negate, xor)

FMV.X.W / FMV.W.X – Move between integer and float registers

Control & Status

Floating-Point Control and Status Register (**FCSR**) – Read/set via CSRs

Includes: Rounding mode (frm), exception flags (fflags), etc.

Chapter 9. Vector operations

Overview of the chapter

Chapter 9 introduces vector processing in RISC-V using the Vector Extension (V-extension). It explains how to perform SIMD-style operations (Single Instruction, Multiple Data), enabling parallel computation for tasks like matrix math, signal processing, or scientific computing. Vector programming is a complex topic and many areas are beyond the scope of this document. More details can be found in section 31 of the unprivileged instruction set manual volume1.

At the time of writing the current version is 20240411 and the document can be found by following the link at https://riscv.org/specifications/ratified/

9.1. Vector system support

The examples shown here were performed on a physical BananaPi BF3 system. The BananaPi has support for vectors[55] as shown by the Linux command below:

```
$ cat /proc/cpuinfo

processor       : 0

hart            : 0

model name      : Spacemit(R) X60

isa                                                                    :

rv64imafdcv_zicbom_zicboz_zicntr_zicond_zicsr_zifencei_zihintpause_zihpm_

zfh_zfhmin_zca_zcd_zba_zbb_zbc_zbs_zkt_zve32f_zve32x_zve64d_zve64f_zve64x

_zvfh_zvfhmin_zvkt_sscofpmf_sstc_svinval_svnapot_svpbmt

mmu             : sv39

uarch           : spacemit,x60

mvendorid       : 0x710

marchid         : 0x8000000058000001

mimpid          : 0x1000000049772200

. . .
```

[55] At the time of writing the default GDB debugger on the BananaPi Bf3 Armbian O/S did not show the vector registers during a debug session. The link https://forum.spacemit.com/t/topic/319?u=alice provided a fix.

```
processor       : 7

hart            : 7

model name      : Spacemit(R) X60

isa                                                                    :

rv64imafdcv_zicbom_zicboz_zicntr_zicond_zicsr_zifencei_zihintpause_zihpm_

zfh_zfhmin_zca_zcd_zba_zbb_zbc_zbs_zkt_zve32f_zve32x_zve64d_zve64f_zve64x

_zvfh_zvfhmin_zvkt_sscofpmf_sstc_svinval_svnapot_svpbmt

mmu             : sv39

uarch           : spacemit,x60

mvendorid       : 0x710

marchid         : 0x8000000058000001

mimpid          : 0x1000000049772200
```

The command string to assemble the vector capable programs used in this chapter is:

```
as  -mno-relax -march=rv64gcv -g -o <filename>.o <filename>.s
```

(indicating that the architecture has *RV64gc*v capability)

Followed by:

```
ld -o <filename> <filename>.o
```

to perform the linking.

If physical hardware is not available there are simulators available[56].

9.2. Vector registers overview

9.2.1. General purpose vector registers

There are 32 vector registers (v0…V31). Vectors can hold scalar values[57] or vector values. The number of *elements*[58] associated with the vector registers is variable and is defined by the total amount of memory available for the vector registers. The number of elements in a vector register is held in the *vector length*

[56] See https://github.com/riscvarchive/riscv-v-spec for references to simulation.

[57] Integers or floating point.

[58] An element is an independent data entity such as the numerator in a division operation.

register. Arithmetic and logical tasks can be performed including multiply/divide, floating-point and shift operations.

Vector registers can be combined into *vector register groups*, allowing a single instruction to operate across multiple vector registers. The *vector length multiplier*, `VLMUL`, represents the number of registers that collectively form a vector register group.

VLMUL has integer values of 1, 2, 4, and 8.

9.2.2. Vector CSR's

There are seven vector associated CSR registers shown in Table 9-1[59]

Table 9-1 Vector CSRs

Address	Privilege level	CSR Name	Meaning
0x008	Unprivileged (Read/Write)	vstart	Vector start position
0x009	Unprivileged (Read/Write)	vxsat	Fixed-point saturate flag
0x00A	Unprivileged (Read/Write)	vxrm	Fixed-point rounding mode
0x00F	Unprivileged (Read/Write)	vcsr	Vector control and status register
0xC20	Unprivileged (Read)	vl	Vector length
0xC21	Unprivileged (Read)	vtype	Vector data type
0XC22	Unprivileged (Read)	vlenb	Vector register byte length

9.2.2.1. VSTART register

The *vector start position register (vstart)* is used to specify the index of the first element to be executed by vector instructions. Listing 9-2 references vector element indices.

9.2.2.2. Vl register

The *vector length register (vl)* contains an unsigned int specifying the number of elements. It is set with the instruction `vset(i)vl(i)` such as `vsetvli t1, t0, e32` where t0 holds the number of elements and e32[60] indicates the elements are 32-bits in size. VL is the number of elements involved in a vector operation.

9.2.2.3. VTYPE register

The *vector data type register* (vtype) indicates the encoding for the *selected element width* (SEW), it occupies bits 5:3 of the vtype register. The SEW bits are defined as shown in Table 9-2. SEW is the bit size of each individual element withing a vector register.

Table 9-2 Vtype SEW bit meaning

VSEW bits			SEW
0	0	0	8

[59] See Vector Extension Programmer's model in volume 1 of the RISC-V instruction set manual for further information

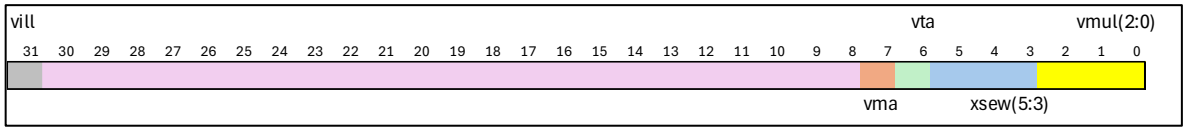
[60] Additionally, e8 corresponds to 8 bits, e16 corresponds to 16 bits and e64 corresponds to 64 bits.

0	0	1	16
0	1	0	32
0	1	1	64

Bits2:0 represents the vector register group multiplier setting collectively termed LMUL. LMUL has mandatory integer values of 1, 2, 4 and 8. Refer to Table 9-3 for bit definitions. The register layout is shown in

Figure 9-1. Fractional values are also supported such as ½ or ¼[61].

Figure 9-1 Vtype register bit fields



The other bitfield definitions (VILL, VMA and VTA) in the vtype register are discussed later in this chapter.

9.2.2.4. VLENB register

The vector byte length (vlenb) register has the value VLEN/8, thus representing values in bytes. It is design-implementation, dependent so could vary by manufacturer. The BananaPi -BF3 (used here) utilizes the SpacemiT K system which has a fixed VLENB value of 32[62].

- ⇨ VLEN=VLENB*8,
- ⇨ VLEN = 32*8 = 256

VLMAX is defined as LMUL * (VLEN/SEW). It represents the maximum length of the number of elements that are involved in a single instruction.

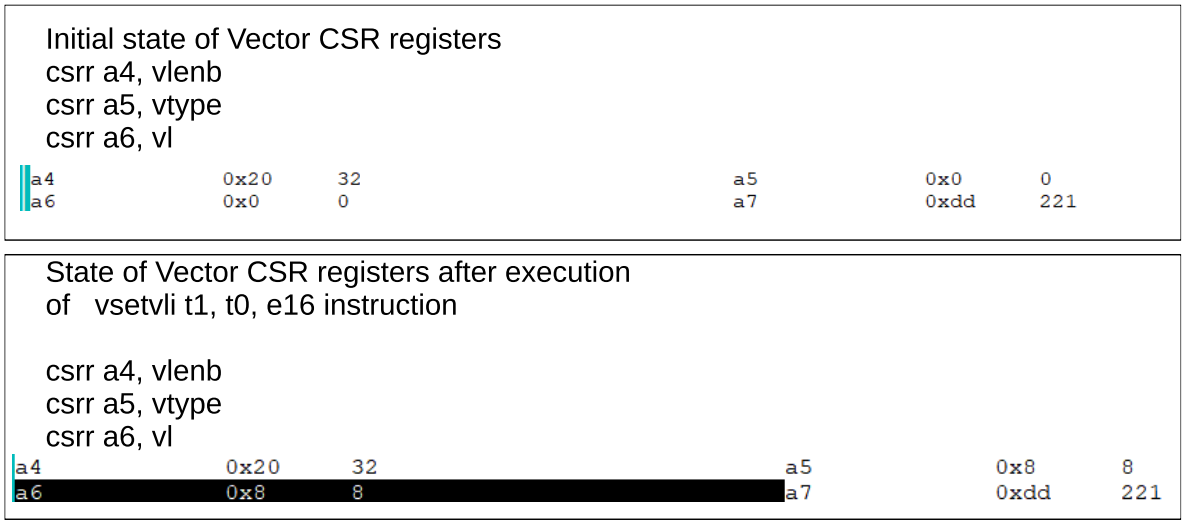
Vector instructions use the `.vv` suffix such as `vadd.vv` to indicate vector operands and the `.vs` suffix to indicate vector and scalar operands. Instructions with three operands would use suffixes such as `.vvv`.

Figure 9-2 shows the vector control and status register state after the `vsetvli` instruction has been executed, here register t0 has been set with the value = 8.

[61] See the specification for further information and rules.

[62] The instruction `csrr rd, vlenb` can be used to return the vlenb value in register rd. The instruction `csr`r is the control and status read comma

Figure 9-2 Using the CSRR instruction to view Vector CSR values


Initial state of Vector CSR registers
csrr a4, vlenb
csrr a5, vtype
csrr a6, vl
a4 0x20 32 a5 0x0 0
a6 0x0 0 a7 0xdd 221
State of Vector CSR registers after execution
of vsetvli t1, t0, e16 instruction
csrr a4, vlenb
csrr a5, vtype
csrr a6, vl
a4 0x20 32 a5 0x8 8
a6 0x8 8 a7 0xdd 221

Note:
Vlenb is unchanged (since vlenb is a constant),
Vtype has changed to 1000 (binary) SEW = 8
Vl has changed to 1000 (binary) there are eight elements in each vector register

The instruction `vsetvli t1,t0, e64`[63] causes the vtype register to change its value to 11000 (binary) and the vl register to change its value to 100 (t0 has been set to 4).

- Vtype (SEW bits = 011b = 64 as the standard element width)
- Vl (100b = 4 elements)
- VLEN/SEW = 256/64 = 4 elements per vector register

To summarize:

VLENB: The amount of bytes in a vector register

VLEN: Related to VLENB, being the amount of bits available in a vector register, must be a power of two

ELEN: The maximum element size for a single vector element, must be a power of two

VL: The number of elements involved in a vector operation

SEW: Defined as the standard element width, (set by `vsetvl` instruction).

LMUL: The vector register grouping value, (2,4 or 8)

[63] The instruction `vseti`**`i`**`vli` allows an immediate value rather than using a register, for example `vsetivli t1,` **`4`**`, e64`

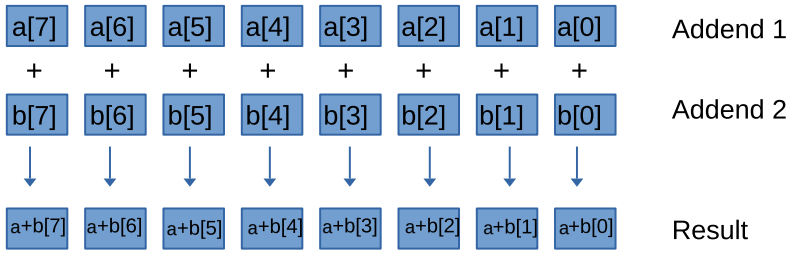
9.3. Vector addition/ subtraction example

The first example adds and then subtracts two vector registers, each register contains a total of 8 elements.

- Vector register1 contains the values
- Vector register2 contains the values
- Vector register3 contains the additive results.
- Vector register4 contains the subtraction results

From the programmer's[64] perspective, the operation takes place in parallel effectively operating on all the elements of two arrays simultaneously - data1[0], data1[1], .. data[i] to data2[0], data2[1], …data2[i] and placing the result in result[0], result[1],..., result[i]. This is shown in Figure 9-3.

Figure 9-3 Simultaneous addition of multiple array elements



Listing 9-1 Vector to vector addition/subtraction

```
# Listing9-1a.s

# RISC-V Vector Addition and subtraction example

# Adds two vectors with 8 elements each

# Each element is 32 bits in size

.text

.global _start

_start:

    # Configure vector parameters

    li t0, 8               # Set vector length (8 elements)

    vsetvli t1, t0, e32    # Set vector length to 8 (t0), element width to

32 bits (e32)
```

[64] This does not *necessarily* mean that the instruction is completed during one hardware clock cycle.

```
    # Load vector data (example values)
    la a0, data1     # Load address of first vector
    la a1, data2     # Load address of second vector
    la a2, addresult # Load address for addition result
    la a3, subresult # Load address for subtraction result
    # Load vectors into vector registers
    vle32.v v1, (a0) # Load first vector into v1
    vle32.v v2, (a1) # Load second vector into v2
# Vector operations take one instruction vv is vector,vector
    vadd.vv v3, v1, v2 # v3 = v1 + v2 (element-wise)
    vsub.vv v4, v1, v2 # v4 = v1- v2
# Store result
    vse32.v v3, (a2)    # Store addition result vector in memory
    vse32.v v4, (a3)    # Store subtraction result vector in memory
# Exit program
    li a7, 93               # Exit syscall number
    li a0, 0                # Exit code 0
    ecall
.data
data1: .word 110, 220, 330, 440, 550, 660, 777, 880     # First vector (8
elements)
data2: .word 100, 200, 300, 400, 500, 600,700,800       # Second vector (8
elements)
addresult:      .word 0, 0, 0, 0, 0, 0, 0, 0            # Addition result
subresult:      .word 0, 0, 0, 0, 0, 0, 0, 0           # Subtraction result
```

Figure 9-4 shows the contents of the vector registers v3 and v4 which hold the results of the vector addition and vector subtraction operations. The content of the vectors is pushed out to memory via the `vse 32.v` instructions and is shown in `GDB` by examining location 0x11170 which is pointed to by the integer register a2.

In total 64 bytes of memory stores the two 32-byte vector registers (v3 and V4). The width of the vector registers was set by `e32` in the `vsetvli t1, t0, e32` instruction[65].

Figure 9-4 GDB showing vector elements

```
      13      # Load vector data (example values)
      14      la a0, data1      # Load address of first vector
      15      la a1, data2      # Load address of second vector
      16      la a2, addresult # Load address for addition result
      17      la a3, subresult # Load address for subtraction result
      18      # Load vectors into vector registers
      19      vle32.v v1, (a0) # Load first vector into v1
      20      vle32.v v2, (a1) # Load second vector into v2
      21
      22      # Vector operations take one instruction vv is vector,vector
      23      vadd.vv v3, v1, v2 # v3 = v1 + v2 (element-wise)
      24      vsub.vv v4, v1, v2 # v4 = v1- v2
      25      # Store result
      26      vse32.v v3, (a2)     # Store addition result vector in memory
      27      vse32.v v4, (a3)     # Store subtraction result vector in memory
      28
      29      # Exit program
  >   30      li a7, 93                # Exit syscall number
      31      li a0, 0                 # Exit code 0
      32      ecall
      33
      34  .data
      35  data1: .word 110, 220, 330, 440, 550, 660, 777, 880      # First vector (8 elements)
      36  data2: .word 100, 200, 300, 400, 500, 600,700,800       # Second vector (8 elements)
      37  addresult:       .word 0, 0, 0, 0, 0, 0, 0, 0            # Addition result
      38  subresult:       .word 0, 0, 0, 0, 0, 0, 0, 0            # Subtraction result
```

```
native process 146953 (regs) In: _start                                    L30    PC: 0x10126
Reading symbols from listing9-1...
(gdb) b 1
Breakpoint 1 at 0x100e8: file listing9-1.s, line 10.
(gdb) run
Starting program: /home/alan/asm/chapter09/listing9-1

Breakpoint 1, _start () at listing9-1.s:10
(gdb) n
(gdb) p $v3.w
$1 = {210, 420, 630, 840, 1050, 1260, 1477, 1680}
(gdb) p $v4.w
$2 = {10, 20, 30, 40, 50, 60, 77, 80}
(gdb) x /16w 0x11170
0x11170:        210     420     630     840
0x11180:        1050    1260    1477    1680
0x11190:        10      20      30      40
0x111a0:        50      60      77      80
```

V3 holds the addition results

V4 holds the subtraction results

Results are stored at the base address pointed to by register a2

This instruction (`vadd.vv v2, v0, v1`) is an example of a **S**ingle **I**nstruction acting on **M**ultiple pieces of **D**ata (*SIMD*).

Disassembly shows:

```
$ objdump -d -M no-aliases listing9-1

listing9-1:     file format elf64-littleriscv
```

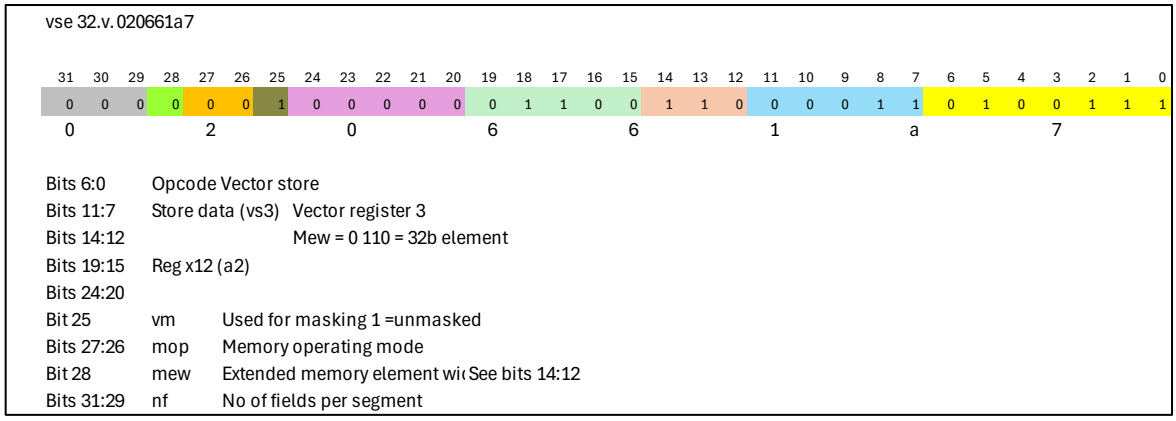
[65] 64-bit width is indicated by *e64*.

```
Disassembly of section .text:


00000000000100e8 <_start>:
   100e8:       42a1             c.li    t0,8
   100ea:       0102f357          vsetvli t1,t0,e32,m1,tu,mu
   100ee:       00001517          auipc   a0,0x1
   100f2:      04250513           addi    a0,a0,66 # 11130 <__DATA_BEGIN__>
   100f6:       00001597               auipc   a1,0x1
   100fa:       05a58593               addi    a1,a1,90 # 11150 <data2>
   100fe:       00001617               auipc   a2,0x1
   10102:      07260613              addi    a2,a2,114 # 11170 <addresult>
   10106:       00001697               auipc   a3,0x1
   1010a:      08a68693              addi    a3,a3,138 # 11190 <subresult>
   1010e:       02056087               vle32.v v1,(a0)
   10112:       0205e107               vle32.v v2,(a1)
   10116:       021101d7               vadd.vv v3,v1,v2
   1011a:       0a110257               vsub.vv v4,v1,v2
   1011e:       020661a7               vse32.v v3,(a2)
   10122:       0206e227               vse32.v v4,(a3)
. . .
```

The instruction `vsetvli t1,t0,e32,m1,tu,mu`

The opcode breakdown for the vector store instruction V3 → Memory vse23.v v3, (a2) is shown in Figure 9-5.

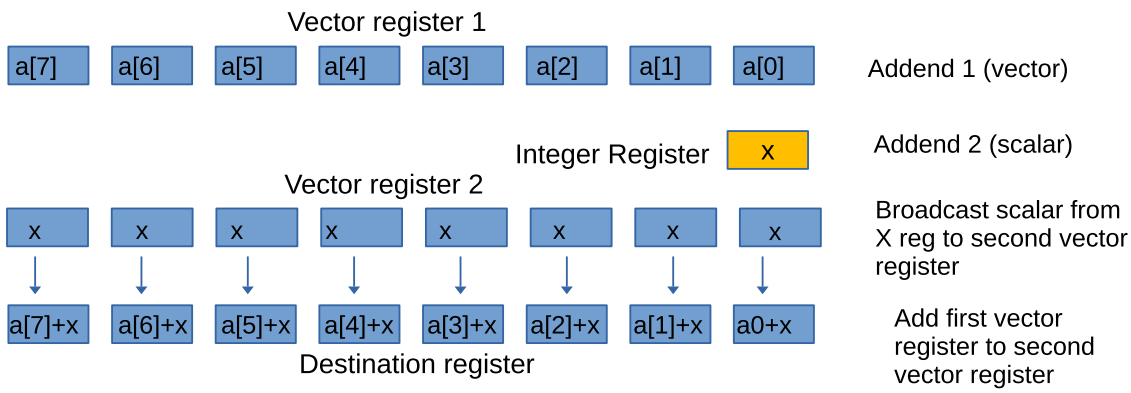
Figure 9-5 Bit field breakdown for vector store instruction


vse 32.v. 020661a7
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 1 1 0 1 0 0 1 1 1
0 2 0 6 6 1 a 7
Bits 6:0 Opcode Vector store
Bits 11:7 Store data (vs3) Vector register 3
Bits 14:12 Mew = 0 110 = 32b element
Bits 19:15 Reg x12 (a2)
Bits 24:20
Bit 25 vm Used for masking 1 =unmasked
Bits 27:26 mop Memory operating mode
Bit 28 mew Extended memory element wi See bits 14:12
Bits 31:29 nf No of fields per segment

9.3.1. Adding a vector and a scalar

The next example adds a vector to a scalar. The scalar value is singular. The first vector register will hold a vector quantity, and the second vector register will hold the scalar. Scalars can be taken from the integer registers or the first element of a vector register. The concept is shown in Figure 9-6.

Figure 9-6 Adding a scalar to all elements of a vector


Vector register 1
a[7] a[6] a[5] a[4] a[3] a[2] a[1] a[0]
Addend 1 (vector)
Integer Register
x
Addend 2 (scalar)
Vector register 2
x x x x x x x x
Broadcast scalar from X reg to second vector register
a[7]+x a[6]+x a[5]+x a[4]+x a[3]+x a[2]+x a[1]+x a0+x
Add first vector register to second vector register
Destination register

The graphic in Figure 9-6 shows a vector register holding an array of eight elements, a scalar quantity is held in an integer register. The content of the integer register is replicated to all elements of a second vector register and finally both vector registers are added together. The code is shown below.

Listing 9-2 Adding a vector and a scalar

```
# Listing 9-2.s

# Vector-Scalar Addition

# v1 = vector (8 elements)

# x10 (a0) = scalar = 15

# Result stored in v2
```

```
# No data section here as the value for the vector registers are generated
# within the program. This program also introduces the concept of stride
# and broadcasting a scalar from an integer register to all elements of
another
# vector register
.text
.global _start

_start:
# Configure vector setting
        li t0, 8              # Set vector length to 8 elements
        vsetvli t0, t0, e32   # 32-bit elements, vl = 8

# Load scalar value into an integer register (a0)
        li a0, 15             # Scalar value = 15

# Generate vector values for v1 rather than obtain them for a .data section
        li t1, 0xffff         # Load integer register t1 with 65535
        vmv.v.x v1, t1       # Set all elements to 0xffff (the value in t1)
        li t1, 11             # Set Stride amount

# vid.v is the vector element index instruction, each element's index is
written from
# 0 to the vector length -1, since vl = 8, 0-7 are written to the dest
register (v0)
        vid.v v0              # indices (0,1,2...) into v0
```

```
    vmul.vx v0, v0, t1    # Multiply indices by stride
    vadd.vv v1, v1, v0    # v1 = [65535, 65546, 65557, 65568, 65579, 65590,
65601, 65612]

# Convert scalar in x register to vector (broadcast)
        vmv.v.x v3, a0        # Broadcast scalar to all elements of v3

# Vector-scalar addition (v2 = v1 + v3)
        vadd.vv v2, v1, v3    # v2[i] = v1[i] + scalar

# Exit (result is in v2)
    li a7, 93             #       Invoke syscall
    ecall
```

By way of introducing new instructions the program does more than simply adding a scalar and vector together. The data was generated using new commands, although it would have been simpler to load the vectors with values defined in the data section this method adds educational value!

Steps 1 through 7 are used to (lengthily) generate the vector content of v1.

The program explanation is as shown.

1. The vector settings are configured for eight elements with a width of 32 bits
2. The code loads a scalar value 0xffff (65535) into the integer register t1
3. The instruction `vmv.v.x v1, t1` moves the value held in the x register t1 to all elements of the vector register v1. Each element in v1 is now {65535, 65535, 65535, 65535, 65535, 65535, 65535, 65535}
4. The *stride* amount is set to 11.
5. The `vid.v v0` instruction writes each elements *index ID* to the destination, the indices are from 0 to the vector length – 1. Since v0 is the destination and the vector length has been set to 8, v0 is now {0,1,2,3,4,5,6,7}.
6. The indices are multiplied by the stride value of 11 (register t1) with the instruction `vmul.vx v0, v0, t1`, giving a result of {0, 11,22,33,44,55,66,77} in vector v0.
7. Vector v0 and vector v1 are added together giving v1 the result {65535, 65546, 65557, 65568, 65579, 65590, 65601, 65612}.

8. The scalar value 15 is replicated (broadcasted) to all elements of v3. Giving v3 the value {15,15,15,15,15,15,15,15}

9. Finally vector v1 and v3 are adding placing the result of {65550, 65561, 65572, 65583, 65594, 65605, 65616, 65527} into vector register v2.

9.3.2. Vector CSR content after execution of Listing 9-2

The CSR register can be shown in `GDB` with the command `info registers vector (I R V)`

```
36          vadd.vv v2, v1, v3    # v2[i] = v1[i] + scalar
37
38  # Exit (result is in v2)
>   39      li a7, 93             #       Invoke syscall
40      ecall
```

```
native process 168292 (regs) In:  start
vstart         0x0        0
vxsat          0x0        0
vxrm           0x0        0
vcsr           0x0        0
vl             0x8        8
vtype          0x10       16
vlenb          0x20       32
```

Vector length = 8

Vector type bits 5:3 = 010 = 32

Vector length in bytes = 32

9.4. Moving elements with vslide

The next example (Listing 9-3) adds individual elements from two vector registers. This is accomplished by extracting the individual elements from the vector registers, placing them in scalar registers and then performing a scalar addition. This is accomplished by the `vslidedown` instruction. For completeness the `vslideup` instruction is included.

The elements are actually moved by the `vslide` instructions which "slides" elements by a number of positions, elements that have been slid out are replaced by zeros. This is similar to shift/rotate operations.

Listing 9-3 Use of vector vslide instructions

```
# Listing9-3.s

# Extract individual elements from vector registers, performs arithmetic,

# placing the result in integer registers


.section .data

vector1: .word 10, 20, 30, 40, 50, 60, 70, 80

vector2: .word 1, 2, 3, 4, 5, 6, 7, 8
```

```
.text
.global _start
_start:
# Load vector from memory
        la a0, vector1
        la a1, vector2
        vsetivli t0, 8, e32  # 8 elements, 32-bit each
        vle32.v v1, (a0)     # v1 = [10,20,30,40,50,60,70,80]
        vle32.v v2, (a1)     # v1 = [1,2,3,4,5,6,7,8]
# Get value at index 2 (30) from vector1
# and value at index 7 (8) from vector2
# Slide and extract
# vslidedown moves an element down a register group
# vslideup moves an element up a register group
# Move down four places, v3 = [50,...], 30 in pole position
     vslidedown.vi v3, v1, 4
# Move down seven places v4 = [8,...], 8 in pole position
    vslidedown.vi v4, v2, 7
# V3 looks like [50, 60, 60, 0, 0, 0, 0, 0]
# V4 looks like [8, 0, 0, 0, 0, 0, 0, 0]
        vmv.x.s t2, v3          # t2 now holds 50 v3[0]
        vmv.x.s t3, v4          # t3 now holds 8 v4[0]
        add t4, t, t3
# Now t4 contains the value 58
```

```
        vslideup.vi v5, v1, 5 # v3 = [...,10,20 30]

        vslideup.vi v6, v2, 6 # v4 = [...,1,2]


    # Exit

    li a7, 93

    ecall
```

Consider the instruction `vslidedown.vi v3, v1, 4` in the listing. Initially vector register 1 contains the eight elements [10, 20, 30, 40, 50, 60, 70 80] and they will be "slid" 4 places downwards (to the left). As the elements are moved leftwards they are replaced from the right by zeros, with the result being placed in vector register v3.

Vector register 1

[10, 20, 30, 40, 50, 60, 70, 80]

[20, 30, 40, 50, 60, 70, 80, 0] Slide down one place

[30, 40, 50, 60, 70, 80, 0, 0] Slide down two places

[40, 50, 60, 70, 70, 0, 0, 0] Slide down three places

[50, 60, 60, 80, 0, 0, 0, 0] Slide down four places

Place this value into vector register 3

`Vslideup` moves the elements rightwards, padding from the left.

9.5. Grouping vector registers

When dealing with certain datasets, it is often not necessary to have 32 vector registers, this might be the case when dealing with comparisons of data held in just two vector registers. Rather than compare eight elements at a time (assuming that 8 is the maximum number of elements having the required data size that can be accommodated by a single vector register the data size) it could be more convenient to process sixteen elements (or more) with each instruction.

By grouping vector registers the model presented to the programmer might be 16 registers each having 16 elements, or 8 registers with 32 elements etc.

Figure 9-6 shows the concept where two 8-element vector registers are combined into one 16-element vector register. Grouping is accomplished by the instruction `vsetivli t0, 16. E32, m2,` here `m2` signifies that the number of groups is 16 (32/2), a value of 4 represents 8 groups and a value of 8 would represent 4 groups. This is shown in Table 9-3.

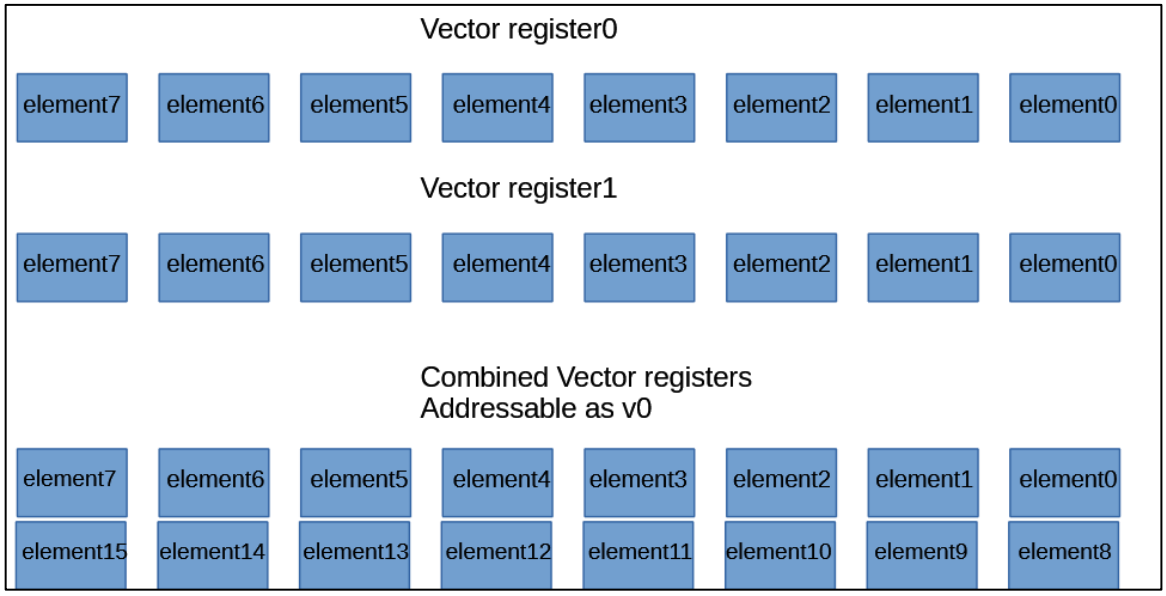
Table 9-3 LMUL and grouping correspondence

Vlmul (2:0)	LMUL	# of groups
000	1	32

001	2	16
010	4	8
011	8	4

The grouped register is addressed as a single operand using the first grouped register so an instruction such as `vle32.v v0, (t0)` would load the data pointed to by t0 into register v0 and v1.

Figure 9-7 Grouping vector registers



Listing 9-4 Shows how to combine vector registers into groups of two.

Listing 9-4 Grouping vector registers

```
# Listing 9-4.s

# Groups two vectors together as one

# V0 and V1 form one group addressed by v0

# V2 and V3 form the second group addressed by V2

# . . .

.section .data

# First vector's contents

dataset1: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
```

```
# Second vector's contents
dataset2: .word 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32
.section .text
.global _start
_start:
# Configure for LMUL=2 with m2 (group 2 registers together)
        vsetivli t0, 16, e32, m2  # 16 elements (2x8), 32-bit, LMUL=2
# Recall LMUL represents the grouping factor
# Load first dataset into v0 (first register in group)
        la t1, dataset1                 #Point to dataset1
        vle32.v v0, (t1)          # Address group by first vector in the
group (v0)
# Load second dataset into the next group)
        la t1, dataset2         #Point to dataset2
# Address group by first vector in the group (v2)
        vle32.v v2, (t1)
# v0-v1: First vector
# v2-v3: Second vector
#Process each group  as single 16-element vector:
# Example :
# Add 2 to all 16 elements of the first grouped vector
# Add 3 to all 16 elements of the second grouped vector
# Use vector integer add instruction
    vadd.vi v0, v0, 2      # Add 2 to first vector
    vadd.vi v2, v2, 3       #Add 3 to second vector
```

```
# Exit

    li a7, 93

    ecall~
```

The instruction `vsetivli t0, 16, e32, m2` includes m2 to set the grouping, previous instances of this instruction did not include an m value which left the default group at 1 → 1 register corresponding to 1 group. The e32 designation is the element size → 32 bits and the preceding number → 16 is the number of elements

Figure 9-8 shows the vector registers before and after the data has been loaded.

Note that a single instruction loads 16 elements, without grouping two instructions would be required – the first instruction to load vector 0 and the second to load vector 1.

Similarly, each `vadd` instruction operates on all of 16 elements of each register pair with one instruction.

Figure 9-8 Loading two vector registers with one instruction

```
        19          vle32.v v0, (t1)         # Address group by first vector in the group (v0)
        20
        21 # Load second dataset into the next group)
  >     22          la t1, dataset2                  #Point to dataset2
        23          vle32.v v2, (t1)         # Address group by first vector in the group (v2)
        24
        25
        26 # v0-v1: First vector
        27 # v2-v3: Second vector
        28
        29      #Process each group  as single 16-element vector:
        30
        31 # Example :
        32 # Add 2 to all 16 elements of the first grouped vector
        33 # Add 3 to all 16 elements of the second grouped vector
        34 # Use vector integer add instruction
        35      vadd.vi v0, v0, 2       # Multiply first vector by 2
        36      vadd.vi v2, v2, 3        # Multiply second vector by 3
        37
        38 # Exit
        39      li a7, 93
        40      ecall
```

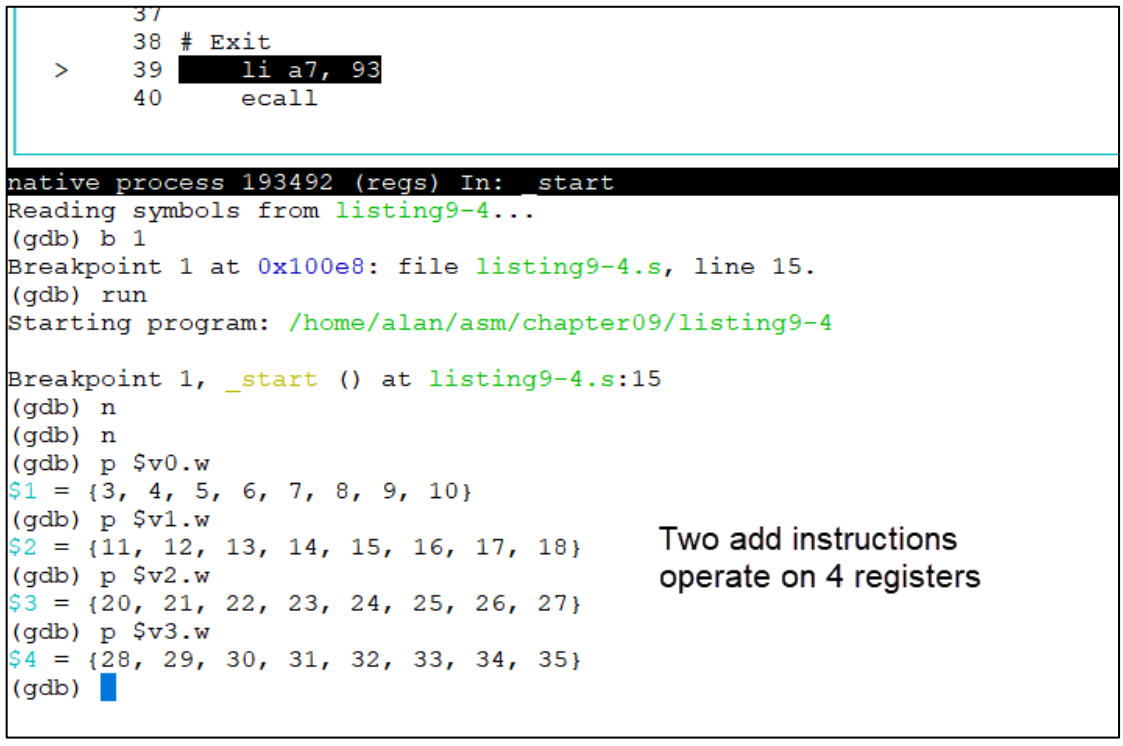
```
ıative process 190216 (regs) In: _start
--Type <RET> for more, q to quit, c to continue without paging--
Reading symbols from listing9-4...
(gdb) b 1
Breakpoint 1 at 0x100e8: file listing9-4.s, line 15.
(gdb) run
Starting program: /home/alan/asm/chapter09/listing9-4

Breakpoint 1, _start () at listing9-4.s:15
(gdb) n
(gdb) n
(gdb) p $v0.w
$1 = {0, 0, 0, 0, 0, 0, 0, 0}
(gdb) p $v1.w
$2 = {0, 0, 0, 0, 0, 0, 0, 0}
(gdb) n
(gdb) p $v0.w
$3 = {1, 2, 3, 4, 5, 6, 7, 8}
(gdb) p $v1.w
$4 = {9, 10, 11, 12, 13, 14, 15, 16}
```

Before vle32.v v0, (t1)

Note one instruction populated both registers

After vle32.v v0, (t1)

Figure 9-9 Operating on two vector registers with a single add instruction


38 # Exit
39 li a7, 93
40 ecall
native process 193492 (regs) In: _start
Reading symbols from listing9-4...
(gdb) b 1
Breakpoint 1 at 0x100e8: file listing9-4.s, line 15.
(gdb) run
Starting program: /home/alan/asm/chapter09/listing9-4
Breakpoint 1, _start () at listing9-4.s:15
(gdb) n
(gdb) n
(gdb) p $v0.w
$1 = {3, 4, 5, 6, 7, 8, 9, 10}
(gdb) p $v1.w
$2 = {11, 12, 13, 14, 15, 16, 17, 18}
(gdb) p $v2.w
$3 = {20, 21, 22, 23, 24, 25, 26, 27}
(gdb) p $v3.w
$4 = {28, 29, 30, 31, 32, 33, 34, 35}
(gdb)
Two add instructions
operate on 4 registers

After the `vsetivli t0, 16, e32, m2` has been executed the CSR registers show the values listed in Figure 9-10. The value 0x11 in the vtype register gives the sew bits (5:3) as 010 and the vmul bits as 001

Figure 9-10 CSR registers after execution of the vsetivli t0, 16, e32, m2 instruction

vl	0x10	16
vtype	0x11	17
vlenb	0x20	32

9.5.1. Masking and merging

RISC-V can merge elements from two vectors based on certain conditions. A *mask* can be used so that a value can be taken from the first source register if a Boolean is true or from the second source register if the Boolean is false with the result going to a destination register. For example a mask consisting of 1,1,0,1,0,1 would take the first two values from rs1, the next value from rs2, the fourth value from rs1, the fifth from rs2 and finally the sixth from rs1. Listing 9-5 shows an example.

Listing 9-5 Use of vmerge instruction

```
# Listing 9-5.s

# Use of mask and vector vmerge instruction


.section .data
```

```
        oddnumbers:   .word 1, 3, 5, 7, 9, 11, 13, 15
        evennumbers:   .word 2, 4, 6, 8, 10, 12, 14, 16
        result: .space 16

.section .text
.global _start
_start:
# Set VL (vector length) to 8 elements
        li      t0, 8
        vsetvli t0, t0, e32, m1

# Load oddnumbers into v1
        la      a1, oddnumbers
        vle32.v v1, (a1)

# Load evennumbers into v2
        la      a2, evennumbers
        vle32.v v2, (a2)

# Set up a mask register: this will select alternate elements odd and even
# Use v0 with binary pattern:  to hold mask bits
        li      t1, 0b1010101010101010
        vmv.v.x v0, t1
        vmerge.vvm v3, v1, v2, v0

# Store the result
```

```
        la      a3, result

        vse32.v v3, (a3)



# Exit

        li      a7, 93

        ecall
```

GDB shows the content of V3 after the merge has been completed.

```
listing9-5.s
      20
      21 # Set up a mask register: thsi will select alternate elements odd and even
      22 # Use v0 with binary pattern:  to hold mask bits
      23         li      t1, 0b1010101010101010
      24         vmv.v.x v0, t1
      25
      26 # vmerge.vvm result into v3 = merge v1 and v2 based on mask v0
      27         vmerge.vvm v3, v1, v2, v0
      28
      29 # Store the result
      30         la      a3, result
      31         vse32.v v3, (a3)
      32
      33 # Exit
 >    34         li      a7, 93
      35         ecall
      36
native process 4218 (regs) In:  _start
Reading symbols from listing9-5...
(gdb) b 1
Breakpoint 1 at 0x100e8: file listing9-5.s, line 10.
(gdb) run
Starting program: /home/alan/asm/chapter09/listing9-5

Breakpoint 1, _start () at listing9-5.s:10
(gdb) n
(gdb) p $v1.w
$1 = {1, 3, 5, 7, 9, 11, 13, 15}
(gdb) p $v2.w
$2 = {2, 4, 6, 8, 10, 12, 14, 16}
(gdb) p $v3.w
$3 = {1, 4, 5, 8, 9, 12, 13, 16}
(gdb) x /16w 0x11168
0x11168:        1       4       5       8
0x11178:        9       12      13      16
```

Memory

9.5.1.1. Other vtype fields

Often in vector processing several elements are unused, for example if 12 elements[66] are processed out of 16, then the unprocessed elements are known as the *tail*. Store/load operations will only work with the processed elements. The remaining tail elements can be set to any value which is known as *tail agnostic* (ta) or they can remain with their previous value which is termed *tail undisturbed* (tu). The policy is set with the `vsetvli` instruction. The unprivileged instruction set manual volume1 now states that the *full form* of the instruction is mandatory and will be required by future code. The full form also includes the mask policy which is *Mask Agnostic* (ma) or *Mask Undisturbed* (mu). An example would be `vsetvli t0, a0, e32, m4, ta, ma.`

The tail policy is set in bit 6 of the vtype register (refer to

---

Figure 9-1) and the mask policy is set in bit 7. Tail elements are set to agnostic when bit 6 is set to 1 and undisturbed when set to 0. There may be occasions where the tail values are important, in this case use tail undisturbed. Use tail agnostic when there is no dependency on the tail elements. In some cases, it may be simpler to just overwrite the tail elements.

Bit 31 is the `vill` bit and normally clear, set if vtype has an illegal value. Bits 8:30 are reserved.

[66] These are the *active* elements

Summary of RISC-V instructions used in chapter 9

Vector Arithmetic Instructions

vadd.vv – Vector + Vector addition

vsub.vv – Vector - Vector subtraction

vadd.vi – Vector + Immediate scalar addition

vmul.vx – Vector × Scalar multiplication

Vector Load/Store Instructions

vle32.v – Load 32-bit elements into a vector

vse32.v – Store 32-bit elements from a vector

Vector Slide Instructions

vslidedown.vi – Slide vector elements down by immediate

vslideup.vi – Slide vector elements up by immediate

Vector Merge and Mask Instructions

vmerge.vvm – Merge vector elements based on mask

vmslt.vv – Set mask if less than (vector-vector comparison)

vmsne.vx – Set mask if not equal (vector-scalar comparison)

Vector Configuration and CSR

vsetvli – Set vector length and configuration

vsetivli – Set vector length with immediate value

vid.v – Generate index vector

vmv.v.x – Move scalar to all vector elements

Register and CSR Usage

- Vector registers used: v0–v31
- CSR-related instructions include:
    - Reading vector control/status via `csrr` (e.g., csrr t0, vtype)

Chapter 10. Spike simulator and Cross compiling

Overview of the chapter

*Chapter 10 focuses on **cross compiling which is** the process of building RISC-V programs on a non-native host machine such as X86-64 to run on a different architecture (RISC-V). The official RISC-V simulator - Spike will be used to run the cross-compiled programs. Spike supports both 32-bit and 64-bit base ISA's and supports vector extensions. There is a proxy kernel PK which provides a run-time environment. Spike also supports debugging operations.*

10.1. Installing the toolchain

Execute the following commands -

```
sudo apt-get install device-tree-compiler autoconf automake autotools-dev curl python3 python3-pip libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev ninja-build git cmake libglib2.0-dev binutils gcc libpthread-stubs0-dev libboost-all-dev
mkdir RISCV
cd RISCV
git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
cd riscv-gnu-toolchain
mkdir build
cd build
../configure --prefix=/opt/riscv --with-arch=rv64gcv --with-abi=lp64d
sudo make
```

10.1.1.1. Set up the path

Modify `.bashrc`

```
vi ~/.bashrc
export PATH=/opt/riscv/bin:$PATH
```

10.2. Installing Spike and PK

10.2.1. Spike installation

```
cd RISCV

git clone https://github.com/riscv-software-src/riscv-isa-sim.git

git clone https://github.com/riscv/riscv-pk.git

cd riscv-isa-sim

mkdir build

cd build

sudo ../configure --prefix=$RISCV --host=riscv64-unknown-elf --with-boost-

libdir=/usr/lib/x86_64-linux-gnu/

make

sudo make install
```

10.2.2. PK installation

```
cd ../../riscv-pk

mkdir build

cd build

../configure --prefix=$RISCV --host=riscv64-unknown-elf

make

sudo make install
```

10.2.3. Testing

Create a small C program

```
vi helloriscv.c

Populate with

#include <stdio.h>


int main() {
```

```
    printf("Hello, RISC-V!\n");
    return 0;
}
```

10.2.4. Cross-compiling C code

```
riscv64-unknown-elf-gcc -march=rv64gcv helloriscv.c
```

Execute the file within the spike environment.

```
alan@vbox:~/RISCV$ spike --isa=rv64gcv pk ./a.out
Hello, RISC-V!
```

Examine the file type

```
readelf -h a.out
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x1014e
  Start of program headers:          64 (bytes into file)
  Start of section headers:          22816 (bytes into file)
  Flags:                             0x5, RVC, double-float ABI
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
```

```
  Number of program headers:         4

  Size of section headers:           64 (bytes)

  Number of section headers:         15

  Section header string table index: 14
```

The program can be transferred over to a native RISC-V host and executed on that host. In the example below the file has been transferred to a Banana Pi BF3 RISC_V native host and then executed.

```
scp a.out 192.168.68.231:

user@192.168.68.231's password:

a.out

$ ./a.out

Hello, RISC-V!
```

Typically, programmers will use spike for initial development and then test their final releases on a native host.

10.2.5. Cross-assembling and linking

The command line for assembling and linking are similar to the commands run on a native host. To differentiate the RISC_V tools from the (in this case) X86-64 tools they are preceded here with `riscv64-unknown—elf-<toolname>.`

Writing the `HelloRiscv` program followed by cross assembling and linking in pure assembly is shown below –

```
$ cat hellorisc.s

# hellorisc.ss

.section .text

.global _start

_start:

li a0, 1 # use a0 for stdout

la a1, message # Load the address of the message text

li a2, 12 # Store the message length

li a7, 64   # Write syscall

ecall
```

```
li a7, 93  # Exit syscall

ecall

.data

message: .ascii "Hello RISCV\n"

$ riscv64-unknown-elf-as -g -o hellorisc.o hellorisc.s

$ riscv64-unknown-elf-ld -o hellorisc hellorisc.o

$ spike --isa=rv64gcv pk hellorisc

Hello RISCV
```

10.2.6. Using objdump

The command to dump the executable is-

```
riscv64-unknown-elf-objdump -d helloriscv.
```

The dissassembled `.text` section looks like:

```
000000000001014e <_start>:

   1014e:   00003197                auipc gp,0x3

   10152:   6ca18193                addi  gp,gp,1738        #        13818
<__global_pointer$>

   10156:   00004517                auipc a0,0x4

   1015a:   86250513                addi  a0,a0,-1950       #        139b8
<__stdio_exit_handler>

   1015e:   00004617                auipc a2,0x4

   10162:   e1a60613                addi  a2,a2,-486 # 13f78 <__BSS_END__>

   10166:   8e09                sub a2,a2,a0

   10168:   4581                li  a1,0

   1016a:   742000ef                jal   108ac <memset>

   1016e:   00001517                auipc a0,0x1

   10172:   95450513                addi  a0,a0,-1708 # 10ac2 <atexit>
```

```
   10176:   c519                    beqz  a0,10184 <_start+0x36>
   10178:   00002517                auipc a0,0x2
   1017c:   ac250513                addi  a0,a0,-1342       #       11c3a
<__libc_fini_array>
   10180:   143000ef                jal   10ac2 <atexit>
   10184:   6c6000ef                jal   1084a <__libc_init_array>
   10188:   4502                lw  a0,0(sp)
   1018a:   002c                addi      a1,sp,8
   1018c:   4601                li  a2,0
   1018e:   04e000ef                jal   101dc <main>
   10192:   b779                j   10120 <exit>
Debugging with Spike
```

Spike has debugging capabilities, it can be invoked by adding `-d` to the command line as follows:

```
spike -d --isa=rv64gcv pk hellorisc
```

Enter “help” to show available actions -

```
(spike) help
Interactive commands:
reg <core> [reg]                # Display [reg] (all if omitted) in <core>
freg <core> <reg>                # Display float <reg> in <core> as hex
fregh <core> <reg>               # Display half precision <reg> in <core>
fregs <core> <reg>               # Display single precision <reg> in <core>
fregd <core> <reg>               # Display double precision <reg> in <core>
vreg <core> [reg]                # Display vector [reg] (all if omitted) in
<core>
pc <core>                        # Show current PC in <core>
insn <core>                       # Show current instruction corresponding
to PC in <core>
```

```
priv <core>                        # Show current privilege level in <core>
mem [core] <hex addr>              # Show contents of virtual memory <hex
addr> in [core] (physical memory <hex addr> if omitted)
str [core] <hex addr>              # Show NUL-terminated C string at virtual
address <hex addr> in [core] (physical address <hex addr> if omitted)
dump                               # Dump physical memory to binary files
mtime                              # Show mtime
mtimecmp <core>                    # Show mtimecmp for <core>
until reg <core> <reg> <val>       # Stop when <reg> in <core> hits <val>
untiln reg <core> <reg> <val>      # Run noisy and stop when <reg> in <core>
hits <val>
until pc <core> <val>              # Stop when PC in <core> hits <val>
untiln pc <core> <val>             # Run noisy and stop when PC in <core> hits
<val>
until insn <core> <val>            # Stop when instruction corresponding to
PC in <core> hits <val>
untiln insn <core> <val>           # Run noisy and stop when instruction
corresponding to PC in <core> hits <val>
until mem [core] <addr> <val>      # Stop when virtual memory <addr> in [core]
(physical address <addr> if omitted) becomes <val>
untiln mem [core] <addr> <val>     # Run noisy and stop when virtual memory
<addr> in [core] (physical address <addr> if omitted) becomes <val>
while reg <core> <reg> <val>       # Run while <reg> in <core> is <val>
while pc <core> <val>              # Run while PC in <core> is <val>
while mem [core] <addr> <val>      # Run while virtual memory <addr> in [core]
(physical memory <addr> if omitted) is <val>
```

```
run [count]                        # Resume noisy execution (until CTRL+C, or
[count] insns)
r [count]                            Alias for run
rs [count]                         # Resume silent execution (until CTRL+C,
or [count] insns)
quit                               # End the simulation
q                                    Alias for quit
help                               # This screen!
h                                    Alias for help
Note: Hitting enter is the same as: run 1
```

The help session uses “core” with many of the commands, here core will have the value 0 representing a single core, a debug session illustrating some of the debug commands follows –

Table 10-1 Spike interactive commands for debugging

Command	Output	Interpretation
pc 0	0x0000000000001000	Program counter at 0x1000
insn 0	0x0000000002028593 addi a1, t0, 32	Current Instruction at Program Counter
run 1	Run 1 line	Advances n instructions
reg 0 t0	0x0000000000001000	Shows the value held in register t0
Reg 0	(spike) reg 0 zero: 0x0000000000000000 ra: 0x000000000(tp: 0x0000000000000000 t0: 0x000000000(s0: 0x0000000000000000 s1: 0x000000000(a2: 0x0000000000000000 a3: 0x000000000(a6: 0x0000000000000000 a7: 0x000000000(s4: 0x0000000000000000 s5: 0x000000000(s8: 0x0000000000000000 s9: 0x000000000(t3: 0x0000000000000000 t4: 0x000000000(	If register is not specified all registers are displayed
until pc 1008	0x0000000000001008	Set a breakpoint at PC=0x1008

`priv 0`	M	Shows current privilege level

The up-arrow key can be used to recall previous commands.

10.3. Further resources

- Information regarding Spike installation can be found at GitHub - riscv-software-src/riscv-isa-sim: Spike, a RISC-V ISA Simulator (https://github.com/riscv-software-src/riscv-isa-sim).
- Installation steps for the RISC-V toolchain can be found at GitHub - riscv-collab/riscv-gnu-toolchain: GNU toolchain for RISC-V, including GCC (https://github.com/riscv-collab/riscv-gnu-toolchain)

Appendix A. GDB Commonly Used Commands

Command	Description	Example
(B)reak	Set breakpoint	“b _start” “b 1”
(D)elete	Delete Breakpoints	“d” followed by “y”
(I)nfo b	Show breakpoints	“i b”
(I)nfo address	Show the location of a Symbol	“i address _start”
(I)nfo files	Show the names of files being debugged	“i files”
(I)nfo (R)egisters	List the integer registers	“i r ”
(I)nfo (R)egister sn	List the content of an individual register	“i r t0”
(I)nfo (R)egisters (V)ector	Shows vector-related registers	“ i r v”
(I)nfo (R)egisters CSR	Shows Control and Status Registers	“I r csr”
P $vn.w	Prints the vector register Vn as groups of words	“p $v2.w”
(I)nfo source	Info about the source file being debugged	“i source”
(I)nfo symbol	Show the symbol at an address	“ i symbol _start”
(I)nfo variables	Shows addresses of variables	“ I variables”
(I)nfo win	Shows windows used in TUI	“i win”
N(ext)	Steps n lines (default is 1) and steps over a sub-routine	“n” “n 3”
S(tep)	Steps n lines (default is 1) and steps into a sub-routine	“s” “s 2”
TUI reg N(ext)	Shows next set of registers	“tui reg n”
x/FMT address	Shows # of memory locations (n), format (f) such as x(hex), d(decimal), f(float), s(string) and size such as b(byte), h(halfword), w (word), g (giant 8 bytes)	X /2xg 0x11100
Up and down arrow	Cycles through commands, use <Ctrl> P(revious) or <Ctrl> N(ext) if using the TUI	

10.3.1.1. Enable TUI

The TUI can be enabled by default by adding the following lines to `~/.gdbinit`

```
Layout split

Layour regs

Set history save on
```

```
Set history filename ~/gdbhistory

Set logging enabled on
```

Other default options are available, refer to the GDB documentation for these!

Appendix B. ASCII Code Table

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0x00	Null character	32	0x20	<SPACE>	64	0x40	@	96	0x60	`
1	0x01	Start of heading	33	0x21	!	65	0x41	A	97	0x61	a
2	0x02	Start of text	34	0x22	”	66	0x42	B	98	0x62	b
3	0x03	End of text	35	0x23	#	67	0x43	C	99	0x63	c
4	0x04	End of transmission	36	0x24	$	68	0x44	D	100	0x64	d
5	0x05	Enquiry	37	0x25	%	69	0x45	E	101	0x65	e
6	0x06	Acknowledgment	38	0x26	&	70	0x46	F	102	0x66	f
7	0x07	Bell	39	0x27	’	71	0x47	G	103	0x67	g
8	0x08	Backspace	40	0x28	(	72	0x48	H	104	0x68	h
9	0x09	Horizontal tab	41	0x29	)	73	0x49	I	105	0x69	i
10	0x0A	Line feed	42	0x2A	*	74	0x4A	J	106	0x6A	j
11	0x0B	Vertical tab	43	0x2B	+	75	0x4B	K	107	0x6B	k
12	0x0C	Form feed	44	0x2C	,	76	0x4C	L	108	0x6C	l
13	0x0D	Carriage return	45	0x2D	-	77	0x4D	M	109	0x6D	m
14	0x0E	Shift out	46	0x2E	.	78	0x4E	N	110	0x6E	n
15	0x0F	Shift in	47	0x2F	/	79	0x4F	O	111	0x6F	o
16	0x10	Data link escape	48	0x30	0	80	0x50	P	112	0x70	p
17	0x11	Device Control 1	49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12	Device Control 2	50	0x32	2	82	0x52	R	114	0x72	r
19	0x13	Device Control 3	51	0x33	3	83	0x53	S	115	0x73	s
20	0x14	Device Control 4	52	0x34	4	84	0x54	T	116	0x74	t
21	0x15	Negative Acknowledgment	53	0x35	5	85	0x55	U	117	0x75	u
22	0x16	Synchronous Idle	54	0x36	6	86	0x56	V	118	0x76	v
23	0x17	End of Transmission Block	55	0x37	7	87	0x57	W	119	0x77	w
24	0x18	Cancel	56	0x38	8	88	0x58	X	120	0x78	x
25	0x19	End of Medium	57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1A	Substitute	58	0x3A	:	90	0x5A	Z	122	0x7A	z
27	0x1B	Escape	59	0x3B	;	91	0x5B	[	123	0x7B	{
28	0x1C	File Separator	60	0x3C	<	92	0x5C	|	124	0x7C	
29	0x1D	Group Separator	61	0x3D	=	93	0x5D	]	125	0x7D	}
30	0x1E	Record Separator	62	0x3E	>	94	0x5E	^	126	0x7E	~
31	0x1F	Unit Separator	63	0x3F	?	95	0x5F	_	127	0x7F	<DEL>

Appendix C. References and Resources

- Green Card Reference sheet https://cs315-f24.cs.usfca.edu/files/RISCVGreenCardv8.pdf
- Ratified Specifications https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications
- RISC-V Summits https://riscv.org/community/risc-v-summits/

INDEX

%%, 7-14

%c, 7-14

%d, 7-14

%e, 7-14

%f, 7-14

%s, 7-14

%u, 7-14

%x, 7-14

%X, 7-14

(V-extension, *9-1*

.global, *2-14*

.include, 6-10

.macro, 6-10

absolute addresses, *3-8*

abstraction, *1-1*

ADDI, 4-7

AND, *1-20*, 4-19

ANDI, 4-19

ASCII, *2-14*

assembler, *2-18*

auipc, *2-9*

BananaPi BPI-F3, *2-27*

Bare metal programming, *2-15*

Base Integer ISA, *2-1*

Basic ASM, 7-7

beq, 5-2

beqz, 5-2

bge, 5-2

bgeu, 5-2

bgez, 5-2

bgt, 5-2

bgtu, 5-2

bgtz, 5-2

biased exponent, *1-17*

Binary Coded Decimal, *1-13*

Binutils, *2-13*

ble, 5-2

bleu, 5-2

blez, 5-2

blt, 5-2

bltu, 5-2

bltz, 5-2

bne, 5-2

bnez, 5-2

B-type, *2-7*

callee, *6-2*

caller, *6-2*

calling routine, *2-4*

compilation stage, *2-18*

CPUlator, *2-36*

cross compiling, ***10-1***

D Double precision, 8-3

debugging, *2-18*

DIV, 4-14

double precision, *1-17*

double-dabble method., *1-15*

Doubleword, *1-10*

ELEN, **9-5**

emulation, *2-27*

encoding, *1-22*

endm., 6-10

Exclusive OR, *1-21*

Executable and Linkable format (ELF)., *2-18*

Extended ASM, 7-7

F Single precision, 8-3

FADD.S, **8-22**

FCLASS.D, **8-22**

FCLASS.S, **8-22**

FCSR, **8-23**

FCVT.D.S / FCVT.S.D, **8-22**

FCVT.W.S / FCVT.S.W, **8-22**

Fedora, *2-27*

FEQ.S / FEQ.D, 8-22

FLD, **8-22**

FLE.S / FLE.D, **8-23**

FLEN, 8-1

floating -point, *1-17*

FLT.S / FLT.D, 8-23

FLW, **8-22**

FMIN.S / FMAX.S, **8-22**

FMUL.S, **8-22**

FMV.X.W / FMV.W.X –, 8-23

Format specifier, 7-14

FSD, **8-22**

FSGNJ.S / FSGNJN.S / FSGNJX.S, 8-23

FSQRT.S, **8-22**

FSUB.S, **8-22**

FSW, **8-22**

funct3, *2-12*

funct5, 8-4

funct7, *2-12*

Functions, 6-1

gcc, 7–3

GDB, *2-21*

GDBinit, *3–19*

global pointer register), *3-17*

H Half precision, 8-3

Halfword, *1-10*

hart, *2-2*

IEEE 754, *1-17*

Instruction Set Architectures (, *2-1*

I-type, *2-7*

JAL, 5–2

JALR, 5-2

J-type, *2-7*

Leaf functions, 6-2

li., *2-10*

LicheePi 4A, *2-27*

linker, *2-18*

linker relaxation, *3-13*

Linker scripts, *2-19*

LMUL, 9-4

lui, *2–9*

lw, *3–3*

Macros, 6-10

make, *2-25*

mask, *9-19*

minuend, *1-10*

mno-relax, 9-2

mno-relax option, *3-19*

MULW, 4-13

nano, *2–14*

Nested functions, 6-2

normalized number, *1-19*

NOT, 4-19

Not-a-number, *1-18*

NVRAM, *1-3*

objdump, 2-17, *2-24*
object file., *2-18*
one's complement,, *1-9*
opcode, *2-12*
optimization, 7-11
OR, *1-20*, 4-19
ORI, 4-19
overflow, *4-10*
plaintext, *2-14*
Pop, 6-1
printf, 7-14
program counter, *2-4*
Program Counter (PC) relative addressing, *3-8*
proxy kernel PK, *10-1*
Pseudo instructions, *2-10*
Pseudocode, *1-2*
Push, 6-1
Q Quad precision, 8-3
QEMU, *2-27*
RARS, *2-37*
Registers, *2-4*
REM, 4-14
REMU, 4-14
RISC, *2-1*
RISC-V, *2-1*
rounding modes, 8-5
R-type, *2-7*
save-temps, 7-3
scalar, 9-2
sections, *2-15*
SEW, **9-5**
signed, *1-8*
significand, *1-17*
Simulators, *2-36*
single precision, *1-17*
sll, 4-16
slli, 4-16
source files, *2-14*
SpacemiT K system, 9-4
Spike, *10-1*
sra, 4-17
srai, 4-17
srl, 4-16
srli, 4-17
stdout, *3-5*
strace, *2-39*
S-type, *2-7*
subtrahend, *1-10*
sw, 2-8
symbol, *2-18*
syntax, *2-18*
Syscalls, *2-14*
tail, *9-22*
tail undisturbed, *9-22*
TUI, *3-19*
two's complement., *1-9*
UDIV(, 4-14
Unconditional branches, *2-11*
unsigned, *1-8*
U-type, *2-7*
vadd.vv, 9-4
variadic function, 7-14
vector byte length, *9-4*
vector data type register, *9-3*
vector length multiplier, *9-3*
vector length register., 9-3

vector register groups, *9-3*

vector start position register, *9-3*

vi, *2-14*

virtualizer, *2-27*

VisionFive2, *2-27*

VLMAX, 9-4

VLMUL, 9-3

VMA, *2-17*

vsetivli t0, 16, e32, m2, 9-18

vsetvli t1,t0, e64, 9-5

vslidedown, 9-13

Vslideup, 9-15

vstart, *9-3*

vtype, 9-3

Word, *1-10*

XLEN, *2-4*

XOR, 4-19

XORI, 4-19